**MASTER EN SCIENCES INFORMATIQUES**

# ArThUR

## Un outil d'aide à la manipulation d'ontologies et de la logique de Markov

Bodart, Axel; Evrard, Keyvin

*Award date:*
2014

*Awarding institution:*
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2013–2014

# ArThUR : Un outil d'aide à la manipulation d'ontologies et de la logique de Markov

Axel Bodart Keyvin Evrard


Maître de stage : Pierre-Yves Schobbens

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Pierre-Yves Schobbens

Co-promoteur : James Ortiz

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

**Résumé**

Les domaines utilisant des programmes d'aide à la décision, et plus particulièrement la médecine, nécessitent à la fois précision et rigueur dans les résultats retournés. Les ontologies permettent de représenter efficacement la connaissance contenue dans un domaine d'application alors que les réseaux logiques de Markov permettent d'inférer sur des règles afin d'en déduire des probabilités d'occurrence. Couplées, ces deux technologies offrent un outil puissant pour calculer efficacement la probabilité d'occurrence d'évènements dans un secteur particulier.

Dans ce mémoire, nous présentons ArThUR [1](Ortiz, Schobbens, Bodart et Evrard, 2014), un programme permettant de manipuler la connaissance contenue dans les ontologies afin de permettre à un moteur reposant sur les réseaux logiques de Markov d'y effectuer des prédictions. Ces prédictions pourront par la suite être analysées pour juger de l'efficacité de ce moteur.

**Mots clés**

Ontologie, Réseau logique de Markov, Alchemy, Tuffy, RockIt, ORTHOGEN

**Abstract**

Domains using intelligence decision support systems, especially medicine, require both precision and rigor for their results returned. Ontologies are used to effectively represent the knowledge contained in a scope while Markov logic networks can infer on a set of rules to derive their likelihood. Coupled together, these two technologies provide a powerful tool to efficiently compute the probability of occurrence of events in a particular sector.

In this thesis, we present ArThUR [1](Ortiz, Schobbens, Bodart et Evrard, 2014), a program to manipulate the knowledge contained in ontologies to allow an engine based on Markov logic networks to make predictions. These predictions can then be analyzed to assess the effectiveness of this engine.

**Keywords**

Ontology, Markov logic network, Alchemy, Tuffy, RockIt, ORTHOGEN

# Remerciements

Nous tenons à remercier toutes les personnes qui nous ont aidés de près ou de loin lors de la réalisation de ce mémoire.

En premier lieu, nous voudrions remercier notre promoteur Pierre-Yves Schobbens, ainsi que notre co-promoteur James Ortiz pour leur disponibilité, leur encadrement et leurs précieux conseils. Sans eux, ce mémoire n'aurait pas pu prendre la forme qu'il a aujourd'hui.

Nous adressons également nos remerciements à Isabelle Daelman et à l'ensemble du secrétariat d'informatique pour nous avoir mis à disposition des locaux lorsque nous en avions besoin.

Nous n'oublions pas nos parents et nos amis pour leur soutien. Nous remercions particulièrement tous nos relecteurs pour avoir pris le temps de lire et de corriger ce mémoire.

Merci à toutes et à tous.

# Glossaire

**Alchemy** Il s'agit d'un des trois moteurs d'inférence probabilistes que nous utilisons au sein de ce mémoire. Ce dernier est notamment utilisé par l'hôpital de Mont-Godinne.. IV, 3, 45, 47, 48, 54, 64, 73, 75, 78, 81–87, 89, 92–94, 98–100, 102–105

**moteur d'inférence probabiliste** Il s'agit d'un programme s'appuyant sur la logique de Markov qui permet d'inférer sur un ensemble de règles et d'évidences afin de calculer des probabilités attachées à ces dernières.. III, 3, 4, 43–47, 49, 51, 52, 54, 59, 62, 65, 78, 92, 93, 98, 100, 102, 103, 107

**ontologie** Il s'agit d'une représentation qui est utilisée pour capturer la connaissance d'un domaine particulier, elle contient la description précise et exacte des concepts du domaine et des relations entre ceux-ci.. III, 2–4, 6, 17, 19–22, 25, 28, 35, 43–47, 49, 51–59, 61, 63–66, 70–74, 78–80, 92, 94, 98, 102, 103, 105, 108

**ORTHOGEN** Il s'agit du nom du projet en collaboration avec l'hôpital de Mont-Godinne avec lequel nous travaillons dans le cadre de notre mémoire.. 1–4, 42–46, 49, 52, 66, 67, 78, 79, 102, 103, 105

**raisonneur de la logique de Markov** Il s'agit d'une autre terminologie que nous utilisons pour parler des moteurs d'inférence probabilistes. 4, 46, 51, 52, 54, 61, 65, 73, 75, 78, 79, 81, 92, 96, 98, 102, 103

**RDBMS** Système de base de données relationnelle : Il s'agit d'un système de gestion de bases de données qui s'appuie sur le modèle relationnel.. 85–87

**RockIt** Il s'agit d'un des trois moteurs d'inférence probabilistes que nous utilisons au sein de ce mémoire.. IV, 3, 45, 48, 53, 54, 62, 64, 65, 73, 75, 78, 89, 91–93, 98–100, 102–106

**Tuffy** Il s'agit d'un des trois moteurs d'inférence probabilistes que nous utilisons au sein de ce mémoire.. IV, 3, 45, 48, 53, 54, 64, 73, 75, 78, 84–89, 91–93, 98–100, 102–106

# Table des matières

# Outline

1. État de l'art
   1.1. Comment peut-on représenter la connaissance en informatique ?
   1.2. Notre quotidien est entouré d'événements incertains.
   1.3. Notre monde est régit par des lois probabilistes.
   1.4. Comment peut-on représenter l'incertitude sous forme de modèle ?
2. Objectifs
   2.1. Aboutissement de notre mémoire pour le projet dans lequel nous avons pris part.
   2.2. Quels problèmes vont être traités dans ce mémoire ?
   2.3. Quelles sont les limites que ce mémoire s'est fixées ?
3. Développement
   3.1. De quoi va traiter le programme que nous avons développé ?
   3.2. Comment avons-nous implémenté ce programme ?
   3.3. Explication de notre module API.
   3.4. La transformation d'une ontologie en règles de la logique du premier ordre.
   3.5. Comment passe-t-on d'un moteur d'inférence probabiliste à un autre tout en transformant la syntaxe utilisée ?
   3.6. Comment l'utilisateur peut-il faire pour manipuler la connaissance d'un domaine d'application.
   3.7. Traduction des règles de la logique du premier ordre vers l'Anglais.
   3.8. Fonction d'historique et de réapplication de changements réalisés sur une ontologie.
   3.9. Intégration de statistiques concernant les temps d'exécution et les résultats obtenus par les différents raisonneurs.
4. Expérimentation
   4.1. Dans quel contexte avons-nous réalisé nos tests ?
   4.2. Quel était le contexte théorique dans lequel nous nous situions ?
   4.3. Analyses des conclusions sur les résultats obtenus.

# Introduction

Depuis ces dernières années, l'aide à la décision a influencé de nombreux domaines. Cette discipline est apparue afin de pallier aux limites de l'être humain en ce qui concerne la connaissance et la mémoire. L'arrivée de l'informatique a amené de nouvelles possibilités d'aide à la décision grâce à la technique qui découlait de cette discipline. Ainsi, au fil du temps, lorsque les sciences informatiques ont commencé à s'imposer, des programmes d'aide à la décision sont apparus sur le marché. Ceux-ci permettent de réaliser des raisonnements automatiques destinés à offrir une série de solutions possibles à l'utilisateur du programme. Le but de ce dernier n'est pourtant pas de substituer entièrement l'homme par la machine, mais simplement de lui fournir une aide sur laquelle il peut s'appuyer en cas de besoin. Ainsi, ce genre de programmes peut, par exemple, grâce à un ensemble de données fournies en entrée, simuler des prévisions sous forme de graphiques. Il peut également offrir une liste de solutions suivant un contexte décrit par l'utilisateur. Cependant, il ne s'agit comme il a déjà été évoqué que d'un support. La décision finale revient toujours à l'utilisateur.

Ainsi, au fil du temps, ces programmes d'aide à la décision se sont implantés dans une multitude de domaines allant de la finance, aux prévisions scientifiques et en passant par la médecine. C'est ce dernier point que nous allons plus particulièrement traiter dans ce mémoire. En effet, ce travail rentre dans le processus de développement du projet ORTHOGEN [2] (Schobbens, De Nizza, Ortiz, Meurisse et Trigaux, 2013). Ce projet vise à fournir un programme d'aide à la décision à l'hôpital de Mont-Godinne, situé en Belgique. Ce programme aura pour but final d'apporter un ensemble de diagnostics possibles suite aux symptômes qu'un médecin décèlera chez un patient. A nouveau, c'est l'expert médical qui aura bien sûr le dernier mot quant à la décision qu'il sera amené à prendre. Cependant, ce programme fournira un lot de probabilités attachées à chaque diagnostic envisageable. Ainsi, il pourra, d'une part, diriger le médecin vers les cas les plus probables suite aux symptômes que rencontre un patient. D'autre part, il permettra également de fournir un ensemble de cas limites, à faibles probabilités, que le professionnel en médecine aurait pu omettre de par leur rareté.

L'intérêt d'un tel outil est justement de soulager le médecin dans sa mémorisation en lui rappelant les scénarios qu'il aurait pu oublier ou ne pas envisager. En effet, particulièrement dans le domaine médical, l'omission d'un cas possible peut amener à des scénarios catastrophiques. Si l'on ne prend pas en compte un symptôme important ou si l'on n'envisage pas une maladie grave car celle-ci semble improbable, le patient pourrait se retrouver dans une situation médicale très délicate. L'utilisation d'ordinateurs dans le milieu de l'aide au diagnostic au sein du domaine médical pourrait permettre également d'adresser l'immensité de la connaissance à ce sujet alors qu'un humain ne pourrait pas forcément y parvenir. Il faut aussi prendre en compte l'apparition de nouvelles maladies, de nouvelles médecines, etc. En effet, les connaissances évoluent et un médecin ne peut pas toujours être au courant des nouvelles trouvailles alors que la mise à jour de la connaissance que traite un ordinateur peut être envisagée. Ici, l'enjeu de vies humaines menacées renforce l'obligation du programme d'aide à la décision d'être le plus précis possible. En effet, particulièrement dans le domaine médical, la rigueur et la précision doivent être des constantes à ne pas bafouer. Un petit écart peut engendrer des dégâts immenses. L'administration d'un traitement qui ne convient pas aux symptômes rencontrés chez une personne malade peut amener des effets secondaires aussi graves que néfastes.

C'est pour cette raison que le projet réalisé par ORTHOGEN se doit d'utiliser les technologies les plus fiables produisant des résultats aussi précis que possible que ce soit au niveau des pathologies éventuelles ou au niveau des probabilités attachées à ces pathologies. C'est pour cette raison que ce projet a choisi de représenter l'ensemble de la connaissance du domaine médical dans lequel il travaille grâce à une ontologie [3](Schobbens, De Nizza, Ortiz et Meurisse, 2013). Les ontologies permettent d'établir une classification des concepts qu'elles décrivent. Cela va permettre, par la suite, de tirer profit de mécanismes de déduction afin de déterminer des relations hiérarchiques entre les différents éléments qui la composent. Grâce à leur sémantique précise et détaillée, elles permettent notamment de capturer des subtilités du monde réel que d'autres représentations ne peuvent pas. Cette technologie va ainsi offrir la possibilité de représenter de manière précise et efficace un domaine spécifique de la médecine. Grâce aux ontologies, il est désormais possible de raisonner sur une base de connaissances. Cependant, pour obtenir des probabilités associées aux déductions effectuées sur cette connaissance, il est nécessaire d'apporter une nouveauté technologique. Les réseaux logiques de Markov ont été choisis pour permettre l'ajout de ces probabilités [4](Matthew Richardson et Pedro Domingos, 2006). Ceux-ci se basent sur un ensemble de règles exprimées dans la logique du premier ordre et infèrent sur cet ensemble afin d'estimer la probabilité de chacune de ces règles. Afin de pouvoir mettre en place les raisonnements effectués sur ces réseaux logiques de Markov, il est nécessaire d'utiliser un outil réalisant cette démarche. Par chance, un assez grand nombre d'entre eux existent déjà

sur le marché actuel : Alchemy [1], Tuffy [2], Markov TheBeast [3], ProbCog [4] ou encore RockIt [5]. L'existence de ces programmes libère le projet de devoir en créer un elle-même. Cependant, cela l'oblige à effectuer un travail de recherche visant à déterminer lequel est le plus performant pour la tâche qu'il doit réaliser. A l'heure actuelle, l'hôpital de Mont-Godinne utilise Alchemy. Il se pourrait toutefois qu'un autre choix s'avère plus intéressant étant donné les critères de précision et de rigueur imposés à ORTHOGEN pour la réalisation de ce projet. Un second problème provient du fait que les ontologies et les réseaux logiques de Markov n'offrent pas de syntaxe commune permettant de les chaîner dans un processus. Or, il est essentiel de réussir à faire communiquer le résultat du premier avec les données d'entrée du second.

C'est dans le but de résoudre ces deux derniers problèmes que notre mémoire vient s'intégrer au sein du projet ORTHOGEN. Nous avons développé un programme, ArThUR [1] (Ortiz et al., 2014), dont un article écrit à son sujet a été accepté à la conférence d'Amantea (Italie) qui se déroulera du 27 au 31 octobre 2014 [6]. Ce programme permet, d'une part, de faire communiquer les ontologies avec les réseaux logiques de Markov de façon automatique et, d'autre part, de choisir lequel des moteurs d'inférence probabilistes – qui sont les cinq outils évoqués plus hauts capables de réaliser des raisonnements sur les réseaux logiques de Markov – est le plus intéressant dans le contexte dans lequel se situera le projet ORTHOGEN.

En dehors de notre contribution à ce projet, notre outil est par ailleurs capable d'apporter son aide dans tous les domaines utilisant les ontologies et les raisonnements probabilistes. Nous avons volontairement veillé à rester le plus général possible dans notre implémentation afin de ne pas nous restreindre à un domaine d'application en particulier, mais à nous élargir vers un maximum d'autres secteurs.

Ce document commencera par expliquer au sein de la section numéro deux les différents éléments théoriques que nous utiliserons tout au long de ce mémoire. Cette section contiendra notamment des explications détaillées concernant les ontologies et les réseaux logiques de Markov, mais également de plus amples explications sur d'autres mécanismes de représentation de concepts ou de calculs de probabilités que nous avons choisi de ne pas intégrer dans notre démarche.

La troisième partie contiendra une explication détaillée de l'outil que nous avons implémenté. Cette partie comportera bien entendu l'explication des transformations réalisées par notre programme afin de permettre le passage d'une

1. http ://alchemy.cs.washington.edu/
2. http ://hazy.cs.wisc.edu/hazy/tuffy/
3. https ://code.google.com/p/thebeast/
4. http ://ias.in.tum.de/research/probcog
5. https ://code.google.com/p/rockit/
6. http ://www.onthemove-conferences.org/index.php/swws2014

ontologie à un réseau logique de Markov. Cependant, en plus de cette fonctionnalité, nous avons ajouté à notre outil une série de facilités permettant à un utilisateur du projet de manipuler la connaissance et les concepts contenus dans l'ontologie de manière facile et rapide. Parmi ces fonctionnalités, nous avons intégré un module de traduction offrant la possibilité aux utilisateurs d'améliorer leurs dialogues avec le personnel médical de l'hôpital. Nous avons également ajouté une fonction d'historique afin de faciliter la manipulation de la connaissance alors même que l'ontologie du projet ORTHOGEN évolue. Nous vous présenterons également une API créée dans un but d'évolution et d'interopérabilité de l'application. Enfin, nous vous présenterons notre module de statistiques dont l'objectif est d'offrir un outil à l'utilisateur sur le choix du moteur d'inférence probabiliste à utiliser dans une situation précise.

C'est à partir de ce dernier module que la partie numéro quatre discutera des différentes expériences réalisées afin de déterminer quel raisonneur de la logique de Markov devrait être utilisé pour le projet ORTHOGEN. Nous commencerons dans cette section par discuter des possibilités théoriques que devrait fournir chaque moteur d'inférence probabiliste. Après cette phase théorique, nous nous pencherons alors sur les expériences que nous avons réalisées et analyserons les résultats obtenus.

La dernière partie sera destinée à apporter les conclusions de ce mémoire. Elle reprendra les deux objectifs principaux de ce document et discutera de leur accomplissement au sein des fonctionnalités que nous vous présenterons.

# Chapitre 1

# État de l'art

## Sommaire



## 1.1 La représentation de la connaissance

Une des branches de l'informatique qui suscite le plus d'intérêt est l'intelligence artificielle [5](Beal). On peut expliquer cela par le fait que cette discipline a pour objectif que les ordinateurs se comportent le plus possible de manière similaire au comportement humain et cela soulève toutes sortes de questions quant aux limites de l'informatique ou encore aux dangers de cette discipline. L'intérêt de l'intelligence artificielle est de tirer profit de la puissance de calcul, de traitement et de la capacité de stockage d'un ordinateur au sein des tâches qui simulent le comportement humain. L'idée est que les ordinateurs sont capables d'exécuter des tâches humaines mais de manière automatique, beaucoup plus rapidement et en général plus précisément. Une des spécialisations de l'intelligence artificielle qui est nécessaire à son fonctionnement est la compréhension du langage humain [5](Beal). Si l'on veut qu'un ordinateur puisse se comporter comme un humain, il faut d'abord qu'il donne du sens à nos mots, qu'il

puisse interpréter nos connaissances pour ensuite pouvoir faire des raisonnements sur base de celles-ci. Avec l'internet qui est la source de connaissances humaines la plus vaste à l'heure actuelle, il est intéressant de trouver un moyen de représenter la connaissance sur un ordinateur de manière à gérer les masses de données existantes ainsi que les caractéristiques inhérentes au langage naturel et ce, de manière efficace. Cette volonté de donner du sens à la connaissance qui se trouve sur l'internet a donné naissance au concept de web sémantique. C'est un concept qui regroupe les méthodes et technologies qui permettent aux machines d'interpréter le sens des connaissances que l'on retrouve sur le web et de pouvoir déduire la signification des objets du domaine. La littérature en ingénierie des connaissances est vaste et nombre de méthodes sont destinées à la représentation des connaissances. Nous avons étudiés les plus pertinentes. Il y a bien évidemment le langage naturel, qui à l'heure actuelle est la représentation la plus utilisée sur le web. Il y a également la représentation logique qui se base sur un système qui regroupe un ensemble de faits et de règles, ces deux représentations ont leurs limites et pour pallier ces limites, la notion d'ontologie a été créée [6](Horridge,2009).

### 1.1.1 Le langage naturel

L'idée la plus simple est de conserver le langage naturel pour représenter la connaissance. Ses avantages sont la facilité d'assimilation pour un utilisateur et la grande expressivité du langage. Mais le problème du langage naturel est qu'il n'a pas de sémantique claire. Dés lors, la compréhension du langage naturel par une machine s'est avérée bien plus compliquée que ce que l'on pensait. Il y a différentes tentatives de traitement automatique du langage naturel [5](Beal) comme la traduction ou encore la reconnaissance vocale mais, durant ces traitements, la machine traite en effet le langage mais elle ne le comprend pas ce qui la rend assez limitée en ce qui concerne ses performances.

Ce sont les caractéristiques de ce langage naturel qui le rendent difficile à interpréter [7](Serres,2003). Tout d'abord le langage naturel n'est pas explicite, c'est à dire que le sens de l'expression n'est pas exprimé clairement, il est sous-entendu. Ces sous-entendus peuvent être liés au contexte de l'expression ou encore déduits grâce à nos connaissances du monde. Cela rend un sous-entendu impossible à prendre en compte pour un ordinateur. Voici un exemple : « Jean est allé faire de la moto. Il est revenu avec une clavicule cassée. » Un ordinateur ne peut saisir l'information implicite de cet énoncé qui dit que Jean est tombé de sa moto et s'est cassé la clavicule. Ensuite, ce qui complique encore plus la tâche c'est que certains termes et certaines expressions peuvent avoir plusieurs sémantiques et que certaines sémantiques peuvent être représentées par plusieurs termes ou plusieurs expressions. C'est ce qu'on appelle respectivement l'ambiguïté et la redondance [7](Serres,2003). Un exemple d'ambiguïté serait : « Je porte la porte » ou encore « Les poules du couvent couvent ». Les termes ont la même forme mais n'ont pas le même sens. « Ma fille a arrêté de lire » et « Laura a renoncé à la lecture » sont deux expressions qui ont le même

sens mais qui contiennent des termes différents, c'est de la redondance dans le langage naturel.

### 1.1.2 Logique du premier ordre

La logique des prédicats ou encore la logique du premier ordre a été élaborée pour pallier les restrictions de la logique propositionnelle. En effet, en logique propositionnelle, il est impossible de parler d'objets, de propriétés d'objets ou encore de mettre en relation des objets. Il n'existe pas non plus de quantificateur sur les objets et le nombre de connecteurs est limité [8](Massé). Il faut autoriser des constructions plus riches si l'on veut pouvoir raisonner sur des assertions. La logique du premier ordre vient donc ajouter la notion de prédicat, de fonction et de quantificateur pour permettre d'être plus expressif avec la logique. Un prédicat prend en entrée une ou plusieurs entités appartenant au domaine du discours et renvoie soit « vrai » soit « faux ». Considérons cet énoncé de la salle d'un cinéma comme exemple : « Toutes les places assises ont un numéro de rangée. Les places à une rangée plus bas offre une meilleure vue de l'écran. Jean est à la première rangée. » Cette énoncé pourrait être traduit de la manière suivante :

- $position(x)$ : fonction d'arité 1 renvoyant le numéro de rangée de x.
- $\forall x \forall y$ : $PlusBas(position(x), position(y)) \Rightarrow MeilleureVue(x, y)$ : ici on retrouve deux prédicats d'arité 2 qui vont permettre d'exprimer que si une personne est a une rangée plus basse qu'une autre, alors elle a une meilleure vue.
- $\forall x$ : $\neg PlusBas(x, position(jean))$ : « Jean est à la première rangée ».

$x$ et $y$ sont des variables dans cette traduction en logique du premier ordre.

La logique du premier ordre est un langage complètement formel, on peut donc déterminer automatiquement si une expression est légale ou non. Un langage du premier ordre $L = (C, P, F)$ est formé [9](Mugnier,2012) :

- d'un ensemble $C$ de constantes (exemple : $jean$) ;
- d'un ensemble $P$ de symboles de relation ou prédicats avec une arité $\geq 0$ (exemple : $PlusBas(x, y)$) ;
- et d'un ensemble $F$ de symboles de fonctions avec une arité $\geq 1$ (exemple : $position(x)$).

La notion d'arité correspond au nombre de variables que l'on retrouve dans un prédicat ou dans une fonction. Pour ajouter la possibilité de parler d'objets comme on en discutait ci-dessus, on a la notion de termes [8](Massé) qui va servir à représenter des objets. En effet, l'ensemble des termes $T$ est le plus petit ensemble tel que :

- toutes les variables et les constantes sont des termes ;
- si $f$ est une fonction de $F$ d'arité $n > 0$ et $t_1$, $t_2$, ..., $t_n$ sont des termes construits sur $L$, alors $f(t_1, t_2, ..., t_n)$ est un terme.

Il faut maintenant parler de la notion de formule logique qui nous intéresse. On obtient une formule en combinant des prédicats et des connecteurs logiques

[8](Massé) :

- une formule $P(t_1, .., t_m)$ est dite atomique si $P$ est un prédicat d'arité m et que $t_1$, ..., $t_m$ sont des termes ;
- on définit une formule par induction :
  - les formules atomiques sont des formules ;
  - si $F$ est une formule, $\neg F$ aussi ;
  - si $F$ et $F'$ sont des formules, $F \vee F'$, $F \Rightarrow F'$, $F \wedge F'$ aussi ;
  - si $F$ est une formule et $x$ une variable , $\forall x \ : F$ et $\exists x \ : F$ sont des formules.

Comme dans toute composition, il existe une priorité a respecter entre les différents connecteurs logiques. Celle-ci est très importante pour l'interprétation ou encore la manipulation des formules logiques. La notion de variable libre et liée peut être définie comme suit[10](Lugiez) :

- si $\varphi$ est un atome alors tout occurence d'une variable $x$ dans $\varphi$ est libre.
- si $\varphi$ est de la forme $\exists x \ \alpha$ ou $\forall x \ \alpha$ alors l'ensemble des variables libres de $\varphi$ est égale à l'ensemble des variables libres de $\alpha$ sans $x$ et toute occurrence libre de $x$ dans $\alpha$ devient liée dans $\varphi$ par le quantificateur introduit.

En ce qui concerne la manipulation de formules, les connecteurs logiques qui permettent de former une formule logique ont des propriétés intéressantes comme l'associativité ou encore la commutativité [11](Jouannaud). Vous trouverez en détail ces propriétés importantes résultant de l'algèbre de Boole en annexe A.1.1. Il existe également la notion d'égalité $\ll = \gg$ définie par [8](Massé) :

- $x = y \Rightarrow f(x) = f(y)$ (formule).
- $x = y \Rightarrow p(x) \Leftrightarrow p(y)$ (prédicat).

Cette définition est valable pour nimporte quelle arité. De plus, l'égalité possède trois propriétés intéressantes : la réflexivité, la symétrie et la transitivité.

Une fois que l'on a la syntaxe, il faut ajouter une couche de sémantique et se poser la question de l'évaluation d'une formule logique. Si on considère un langage du premier ordre $L = (C, P, F)$, une interprétation de $I$ de $L$ [9](Mugnier,2012) est constituée d'un ensemble $D$ non vide appelé domaine et d'une définition du sens des symboles de $L$ sous forme d'application dans $D$ ou de relations sur $D$ [10](Lugiez) :

- Étape 1 : choisir un domaine $D$ qui est un ensemble non vide tel que $\forall c \in C$, $I(c)$ est un élément de $D$.
- Étape 2 : donner un sens aux fonctions. Pour toute fonction $f \in F$ d'arité $n$ on associe une fonction $I(f)$ de $D^n \to D$.
- Étape 3 : donner un sens aux prédicats. Pour tout prédicat $p \in P$ d'arité $n$ on associe une relation $I(p)$ de $D^n \to \mathbb{B}$. Si $n = 0$ alors $I(p)$ est soit vrai soit faux.
- Étape 4 : donner une valeur de vérité à une formule logique étant donné une interprétation $I$. Il faut en premier lieu définir une affectation $A : V \to D_I$

où $V$ est l'ensemble des variables. L'affectation a pour but de donner une valeur aux variables libres dans les formules. On a que $A[x/c]$ représente l'affectation $A$ au sein de laquelle les occurences de la variable $x$ sont remplacées par la valeur $c$. La sémantique de la formule logique $\varphi$ selon une interprétation $I$ et une affectation $A$, notée $[\![\varphi]\!]^{I,A}$, respecte les règles suivantes [8](Massé). Notez que les opérateurs à gauche sont juste des symboles, alors qu'à droite ils représentent un calcul de valeurs de vérités :

- $[\![x]\!]^{I,A} = A(x)$ ;
- $[\![f(t_1, ..., t_n)]\!]^{I,A} = I(f)([\![t_1]\!]^{I,A}, ..., [\![t_n]\!]^{I,A})$ ;
- $[\![p(t_1, ..., t_n)]\!]^{I,A} = I(p)([\![t_1]\!]^{I,A}, ..., [\![t_n]\!]^{I,A})$ ;
- $[\![\neg\varphi]\!]^{I,A} = \neg[\![\varphi]\!]^{I,A}$ ;
- $[\![\varphi \vee \varphi']\!]^{I,A} = [\![\varphi]\!]^{I,A} \vee [\![\varphi']\!]^{I,A}$ ;
- $[\![\varphi \wedge \varphi']\!]^{I,A} = [\![\varphi]\!]^{I,A} \wedge [\![\varphi']\!]^{I,A}$ ;
- $[\![\varphi \Rightarrow \varphi']\!]^{I,A} = [\![\varphi]\!]^{I,A} \Rightarrow [\![\varphi']\!]^{I,A}$ ;
- $[\![\forall x \ : \varphi]\!]^{I,A} = \bigwedge_{c \in D_I} [\![\varphi]\!]^{I,A[x/c]}$ ;
- $[\![\exists x \ : \varphi]\!]^{I,A} = \bigvee_{c \in D_I} [\![\varphi]\!]^{I,A[x/c]}$ .

Un terme instancié est un terme qui ne contient plus de variable et un prédicat instancié est un prédicat qui ne contient que des termes instanciés. Quand on parle d'un monde possible, on parle de l'assignation d'une valeur de vérité à l'ensemble des prédicats instanciés possibles de Herbrand. Pour pouvoir évaluer la valeur de vérité de la formule logique résultante des connexions, il faut connaître les tables de vérités des différents connecteurs logiques [11](Jouannaud). Pour ce faire, vous pouvez trouver les tables de vérité de l'algèbre de Boole dans la figure 1.1.

FIGURE 1.1 – Tableau de vérité

| **Tableau de vérité** | | | | |
|---|---|---|---|---|
| A | B | $A \wedge B$ | $A \vee B$ | $A \Rightarrow B$ |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 |

Une manipulation de formule qu'il est intéressant d'expliquer est la mise en forme normale conjonctive [9](Mugnier,2012). C'est à dire que l'on transforme une formule logique en une conjonction de disjonctions. Le résultat de la transformation préserve la satisfiabilité de la formule. La forme normale conjonctive est importante car elle va permettre d'énormes facilités d'utilisation de la formule dû à la simplicité de cette nouvelle forme comme par exemple pour l'inférence automatisée. Toute formule de la logique du premier ordre bien formée possède une forme normale conjonctive. Les étapes de la transformation sont automatisables :

- Étape 1 : éliminer les implications et les équivalences :
  - remplacer $A \Leftrightarrow B$ par $(A \Rightarrow B)$ et $(B \Rightarrow A)$ ;
  - remplacer $A \Rightarrow B$ par $\neg A \vee B$.
- Étape 2 : déplacer les négations le plus à l'intérieur ;
  - $\neg(\forall x : \text{p})$ devient $\exists x \ : \neg p$ ;
  - $\neg(\exists x : \text{p})$ devient $\forall x \ : \neg p$ ;
  - $\neg(A \wedge B)$ devient $\neg A \vee \neg B$ ;
  - $\neg(A \vee B)$ devient $\neg A \wedge \neg B$ ;
  - $\neg\neg A$ devient A.
- Étape 3 : mise en forme prenexe [10](Lugiez) ;
  - en premier lieu, il faut standardiser les variables liées en les renommant de façon à ce qu'aucune variable n'ait à la fois une occurrence libre et une occurrence liée et que toutes les occurrences liées d'une variable le soient par le même quantificateur ;
  - ensuite, il faut mettre tous les quantificateurs en tête de formule. Une formule prenexe est une formule de la forme : $(\forall|\exists)^* x_1, ..., x_n f$ où $f$ est une formule ne contenant pas de quantificateur.
- Étape 4 : appliquer la skolémisation [10](Lugiez). Cela consiste à éliminer toutes les occurrences de quantificateurs existentiels en utilisant de nouveaux symboles de fonctions (un par quantificateur existentiel). Si on considère $\forall x \exists y \ : P(x, y)$ on peut se dire qu'il existe une fonction $f(x)$ qui donne le bon y pour chaque $x$. On peut dès lors supprimer le quantificateur existentiel et obtenir $\forall x \ : P(x, f(x))$. Chaque occurrence d'une variable $y$ quantifiée existentiellement est remplacée par un terme $f(x_1, ..., x_n)$ où $x_1, ..., x_n$ sont les variables quantifiées universellement dont le quantificateur se trouve avant celui de $y$.
  - $\exists x \ : Exemple(x)$ devient $Exemple(C)$ où $C$ est une nouvelle constante ;
  - $\forall x, y \exists z \ : P(x, y, z)$ devient $\forall x, y \ : P(x, y, f(x, y))$ où $f$ est une nouvelle fonction.

  Il faut bien évidement choisir des noms de nouvelles fonctions qui n'existent pas encore ;
- Étape 5 : supprimer les quantificateurs universels ;
- Étape 6 : distribuer les $\wedge$ sur les $\vee$ .
  - $(A \wedge B) \vee C$ devient $(A \vee C) \wedge (B \vee C)$.

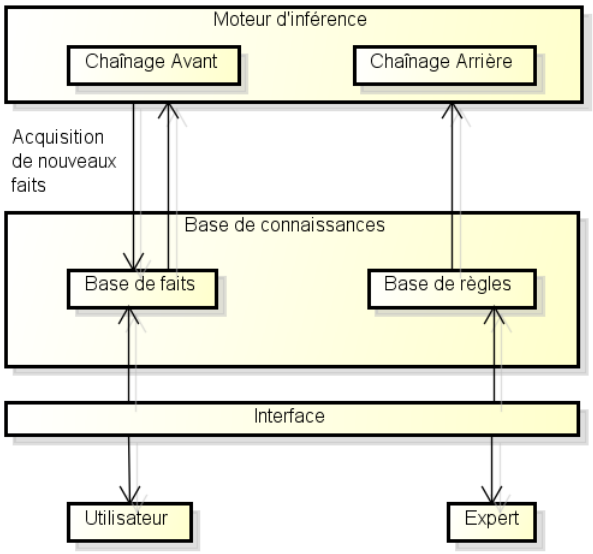
Ce qui nous intéresse avec la logique du premier ordre, c'est la possibilité de représenter la connaissance qu'elle nous offre. En effet, un ensemble de formules de la logique du premier ordre peut être considéré comme une base de connaissances du premier ordre. Les formules logiques d'une base de connaissances sont conjointes implicitement et peuvent donc être vues comme une seule formule de grande taille. Une formule est dite satisfiable [8](Massé) si et seulement si il existe au moins une interprétation dans laquelle elle est vraie. L'inférence basique en logique du premier ordre consiste à déterminer si une base de connaissances implique une formule $F$. On regarde si la formule $F$ est vraie dans tous les mondes où la base de connaissances est vraie. Le procédé logique le plus

souvent utilisé pour faire cela est le procédé de réfutation qui consiste à nier la proposition. On prend l'union de la base de connaissance avec la négation de la formule $F$ et si cette union est insatisfiable alors la base de connaissances satisfait $F$.

### 1.1.3 Système à base de faits et de règles

Un système à base de faits et de règles ou encore système expert [12](Pain et Morignot,2013) est un système qui comprend une base de faits et une base de règles ainsi qu'un moteur d'inférence qui va permettre d'essayer de reproduire des raisonnements comme le ferait un expert en la matière. En figure 1.2, on peut voir l'illustration d'un système expert.

FIGURE 1.2 – Système expert




La base de faits va plutôt contenir les informations du domaine concerné par le problème alors que la base de règles va modéliser les différents liens et relations entre les faits. Les deux forment la base de connaissances qui être assez grande. Le langage utilisé pour représenter les faits et les règles est la logique formelle du premier ordre présentée ci-dessus. Un langage de programmation déclarative qui est proche de la logique du premier ordre est Prolog [13](Jacquet). Celui-ci possède des mécanismes d'inférence intéressants. Nous utiliserons ce langage dans nos exemples. Voici l'exemple d'une description d'une journée d'excursion :

- Une excursion se compose de trois parties :
  - soit d'une promenade, d'une activité principale et de manger un morceau de cake ;

  - soit d'une activité principale, d'une heure de détente et d'une dégustation de jus de fruit.
- Une activité principale, c'est :
  - soit un atelier artistique ;
  - soit une visite du musée.
- Comme promenade, il y a 5, 10 et 15 km.
- La piscine est la seule activité de détente.
- Il y a uniquement la peinture comme atelier.
- Le choix de jus de fruit est jus tropical ou jus multi-fruit.

Tout d'abord, il faut traduire l'énoncé en logique du premier ordre :

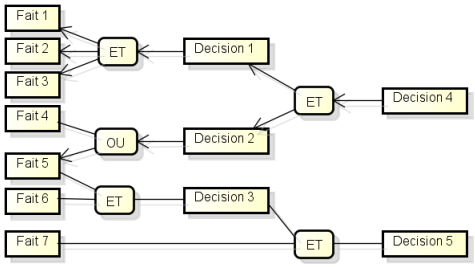
- $\forall x, y, x \ : excursion(x, y, z) \Leftrightarrow (promenade(x) \wedge activite\_principale(y) \wedge z = cake) \vee (activite\_principale(x) \wedge detente(y) \wedge degustation\_jus(z))$
- $\forall x \ : activite\_principale(x) \Leftrightarrow atelier\_artistique(x)) \vee x = visite\_musee$
- $\forall x \ : promenade(x) \Leftrightarrow x = 5km \vee x = 10km \vee x = 15km$
- $\forall x \ : detente(x) \Leftrightarrow x = piscine$
- $\forall x \ : degustation\_jus(x) \Leftrightarrow x = tropical \vee x = multi\_fruit$
- $\forall x \ : atelier\_artistique(x) \Leftrightarrow x = peinture$

Et ensuite, il ne reste plus qu'à traduire cela en programme logique pour obtenir notre base de connaissances. Notez que les équivalences $\ll \Leftrightarrow \gg$ sont traduites par des implications $\ll \leftarrow \gg$, la complétion de prédicat est supposée :

1. $excursion(x, y, cake) \leftarrow promenade(x), activite\_principale(y).$
2. $excursion(x, y, z) \leftarrow activite\_principale(x), detente(y), degustation\_jus(z).$
3. $activite\_principale(x) \leftarrow atelier\_artistique(x).$
4. $activite\_principale(visite\_musee).$
5. $promenade(5km).$
6. $promenade(10km).$
7. $promenade(15km).$
8. $detente(piscine).$
9. $degustation\_jus(tropical).$
10. $degustation\_jus(multi\_fruit).$
11. $atelier\_artistique(peinture).$

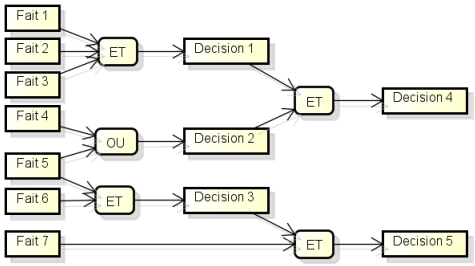
Les raisonnements obtenus par le moteur d'inférence rend le système capable de résoudre des problèmes en manipulant les connaissances et en faisant des déductions. Durant les déductions, le système est capable de déduire de nouvelles connaissances, de nouveaux faits en vue rendre la base de faits plus complète [13](Jacquet). Le but d'un système expert est d'arriver à déclencher des déductions faites par le moteur d'inférence qui va interpréter les règles et ce en vue d'arriver à obtenir une solution à un problème considéré. Il existe deux stratégies de raisonnement différentes [12](Pain et Morignot,2013). Le chaînage arrière (voir figure 1.3) est guidé par l'objectif à atteindre : on part de l'objectif à atteindre et on reconstruit un arbre de preuve pour remonter aux faits. La

FIGURE 1.3 – Chaînage arrière




deuxième stratégie est le chaînage avant (voir figure 1.4) où on démarre des faits et on essaye de produire le plus de nouveaux faits possibles en combinant les faits et règles existants jusqu'à, on l'espère, trouver des faits répondant à l'objectif.

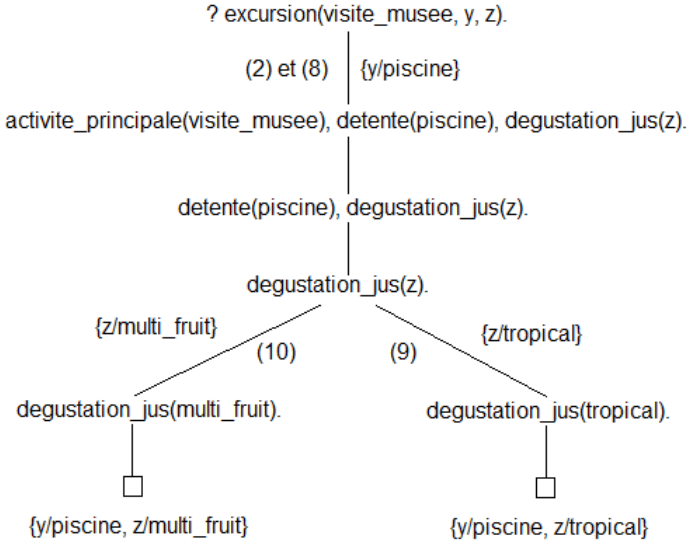
FIGURE 1.4 – Chaînage avant




Pour activer cette inférence, il faut poser une question au système expert sous forme de formule formelle à démontrer à travers les déductions [13](Jacquet). Et en résultat de cette requête le système va soit dire que celle-ci est insatisfiable soit retourner l'ensemble des solutions possibles. Une solution possible correspond à une instanciation des variables de la requête. Par exemple, on pourrait se demander qu'est-ce qu'il faut en deuxième et troisième parties de journée pour obtenir un excursion sachant que, en première partie de journée, une visite au musée est prévue. En figure 1.5, on a la déduction faite par le moteur d'inférence utilisant le chaînage arrière [13](Jacquet).

Les problèmes que l'on doit considérer dans notre monde sont souvent basés sur des domaines assez énormes. Il s'avère que quand on utilise un domaine très vaste avec un système expert, le comportement de celui est assez « fragile » aux frontières du domaine. Les systèmes experts acceptent des domaines assez étroits alors que l'on veut pouvoir gérer des domaines assez larges lors des raisonnements. Un autre problème de cette représentation de la connais-

FIGURE 1.5 – Exemple de chaînage arrière



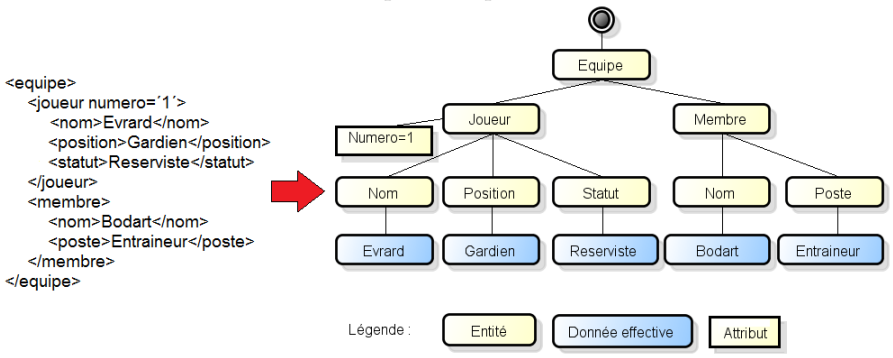

sance c'est qu'il manque des aspects pour pouvoir s'adresser à notre monde de manière complète. Il n'y a pas de représentation du possible ou du nécessaire, tout comme il n'y a pas de représentation du temps [13](Jacquet).

### 1.1.4 Les langages du web sémantique

Depuis quelques années, le web est devenu la source d'informations la plus importante au monde et cela va en croissant. Mais l'on sait que la plupart des services fournis sur le web sont effectués par des ordinateurs. Jusqu'à présent ces ordinateurs traitent l'information sans la comprendre et sont par conséquent moins efficaces dans le traitement de celle-ci. L'idée de base du web sémantique est de donner une certaine structure à l'information pour lui donner une sémantique, une sémantique que les ordinateurs pourraient interpréter [14](Wu). Dès lors, ils pourraient comprendre la connaissance se trouvant sur le web et devenir plus précis et efficaces dans la manipulation de cette connaissance. L'attrait du web sémantique est l'idée de pouvoir arriver à tirer profit de cette quantité énorme d'informations et ce de manière automatique. Pour arriver à cela, il faut une identification et un accès aux ressources du web de façon simple et efficace. Cela nécessite également des langages pour décrire le contenu des documents web ainsi qu'un mécanisme d'inférence pour déduire des nouvelles connaissances à partir de ces documents. C'est dû à cette nécessité que différents standards ont été créés (RDF, RDFS, OWL). Cet intérêt pour le web sémantique a permis de faire des avancées dans les technologies de représentation de la connaissance contenue sur le web mais ces avancements peuvent totalement être utilisés en dehors du web pour représenter de la connaissance provenant de sources différentes.

**eXtensible Markup Language** Le formalisme XML est une structure qui va permettre de définir des langages avec balisage. Chaque langage qui en résulte possède ses propres balises [15](Thiran). Les différentes balises sont imbriquées pour indiquer qu'un concept est contenu dans un autre. Dès lors, cette imbrication peut être transformée de manière équivalente en une structure arborescente avec différents types de nœuds [15](Thiran). Les feuilles de l'arbre contiennent les données effectives sous forme de texte. Il y a des nœuds qui représentent des éléments du domaine, ceux-ci sont ordonnés, et des nœuds qui représentent les attributs des éléments. Une des limitations principale de ce formalisme est qu'on ne peut donner aucune information sur la structure des données car il est essentiellement utile à l'échange de données, ce qui rend possible une certaine ambiguïté au niveau des noms d'éléments [15](Thiran). En figure 1.6, on trouve un exemple de réprésentation XML sous forme textuelle et sous forme d'arbre.

FIGURE 1.6 – Exemple de représentations en XML

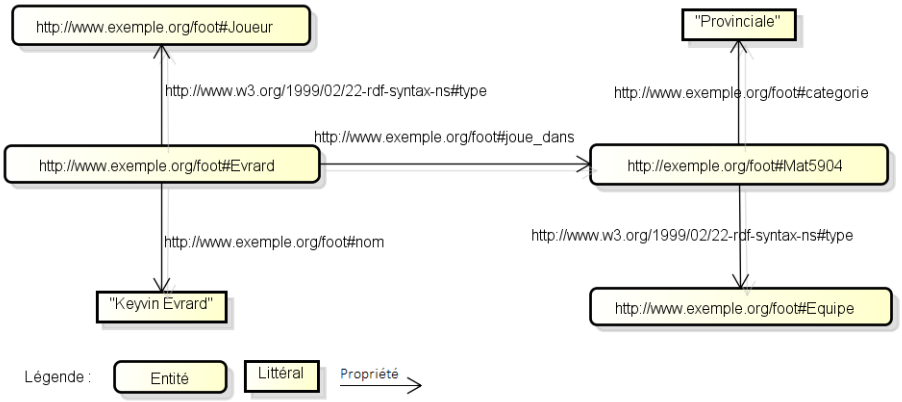



**Resource Description Framework** Le « Resource Description Framework » (RDF) définit une structure qui va permettre de contenir, interpréter et échanger des métadonnées. Ce modèle a été défini dans le but premier de partager des ressources provenant du web d'ailleurs c'est le langage de base du web sémantique [16](Lawarrée,2010) mais il peut également être utilisé pour toutes sortes de données. RDF est une application du langage XML qui va ajouter des contraintes sur la structure pour empêcher les problèmes d'ambiguïté dans l'interprétation de la sémantique du formalisme XML [17](Thiran). RDF va permettre de représenter la connaissance sous la forme d'un multi-graphe dirigé où chaque arc est labellisé. Ces labels sont appelés « propriétés ». On retrouve deux types de nœuds dans ce graphe, les nœuds « entités » qui représentent les entités du domaine et les nœuds « littéraux » qui représentent des valeurs particulières du domaine. Ces derniers ne peuvent avoir que des arcs entrant [17](Thiran).

Pour éviter toute ambiguïté dans la sémantique du graphe, chaque entité et chaque propriété sont identifiées par un identifiant uniforme de ressource

(URI) [18](Heflin). Un URI est une chaîne de caractères qui permet d'identifier une ressource, celle-ci a une syntaxe qui respecte une norme bien précise. La modélisation de la connaissance nécessite souvent de représenter l'inclusion d'où l'existence de différents types de conteneurs dans la syntaxe du langage. Il y a les ensembles, les séquences et les alternatives [17](Thiran). Le premier est un groupe non ordonné alors que dans le deuxième l'ordre a de l'importance. L'alternative représente le ou exclusif, ce qui permet le choix d'un seul élément dans l'ensemble. Si l'on veut représenter un ensemble fermé, il faut alors utiliser la notion de collection aussi présente dans la syntaxe et qui respecte la structure d'une liste. Voici, en figure 1.7, un exemple de graphe RDF.

FIGURE 1.7 – Exemple de graphe RDF avec URI




Le domaine modélisé en RDF peut également être représenté comme un ensemble de triplets de la forme sujet, propriété, objet [18](Heflin). Chaque arc du graphe peut être transformé en un triplet. Le sujet est une entité, la propriété appliquée à cette entité et l'objet est la valeur de la propriété qui peut être soit un littéral soit une autre entité. Dès lors, si on veut facilement stocker la représentation RDF, il suffit d'enregistrer l'ensemble des triplets correspondant. Si on reprend notre exemple ci-dessus, en s'abstrayant des URI pour l'exemple, on obtient l'ensemble de triplets dans la figure 1.8.

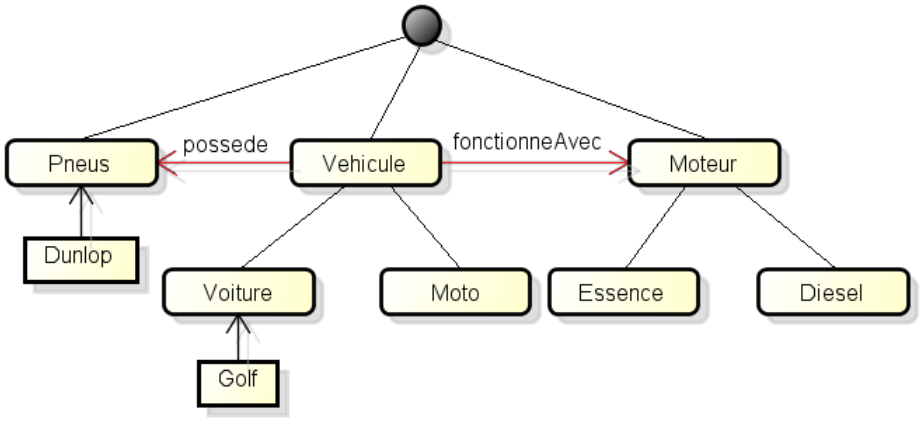
**Resource Description Framework Schema** RDF est un langage permettant de représenter des entités et propriétés d'un domaine au sein d'une modélisation qui a une sémantique bien spécifique. Ce domaine en question qui contient des concepts et des relations peut être modélisé de manière plus abstraite et ce grâce à un autre langage appelé « Resource Description Framework Schema » qui permet de spécifier un schéma qui décrit les métadonnées du domaine [16](Lawarrée,2010). C'est ici que l'on va retrouver les contraintes sur la structure du modèle. RDFS sert essentiellement à définir le sens des propriétés [18](Heflin) en déterminant, par exemple, les valeurs que peuvent prendre les

FIGURE 1.8 – Ensemble de triplets

| Sujet | Propriété | Objet |
|---|---|---|
| Evrard | type | Joueur |
| Evrard | nom | « Keyvin Evrard » |
| Evrard | joue_dans | Mat5904 |
| Mat5904 | type | Equipe |
| Mat5904 | categorie | « Provinciale » |

différentes propriétés ou encore en établissant une hiérarchie entre elles. En figure 1.9, on voit un exemple de graphe RDFS.

FIGURE 1.9 – Exemple de graphe RDFS




Ce langage permet également de définir des classes d'entité avec *rdfs :Class* et à partir de cette notion de classe, il est possible d'établir une hiérarchie entre les entités en utilisant la propriété *rdfs :SubClassOf* [17](Thiran). On peut trouver un exemple de l'extension RDFS en figure 1.10 où *rdfs :domain* permet de définir le type d'entité qui doit être associé au sujet de la propriété et *rdfs :range* permet de définir quelles valeurs peut prendre la propriété [17](Thiran).

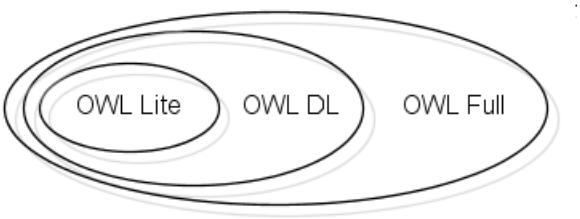
**Web Ontology Language** Un autre standard défini par la suite est le « Web Ontology Language » (OWL) qui est une extension des deux standards RDF et RDFS. L'idée était que ces deux derniers manquaient d'expressivité et de mécanismes de raisonnement. OWL possède un vocabulaire plus expressif, ce qui permet une description plus précise des entités du domaine et de leurs relations, ce qu'on appelle une ontologie, et tire profit des mécanismes d'inférence de la logique des prédicats et en particulier des logiques de description. Pour

FIGURE 1.10 – Exemple d'utilisation du langage RDFS

```
<rdfs:Class rdf:about="http://www.exemple.org/garage#Voiture">
        <rdfs:subClassOf rdf:about="http://www.exemple.org/garage#Vehicule/>
</rdfs:Class>
<rdfs:Class rdf:about="http://www.exemple.org/garage#Moto">
        <rdfs:subClassOf rdf:about="http://www.exemple.org/garage#Vehicule/>
</rdfs:Class>
<rdfs:Class rdf:about="http://www.exemple.org/garage#Essence">
        <rdfs:subClassOf rdf:about="http://www.exemple.org/garage#Moteur/>
</rdfs:Class>
<rdfs:Class rdf:about="http://www.exemple.org/garage#Diesel">
         <rdfs:subClassOf rdf:about="http://www.exemple.org/garage#Moteur/>
</rdfs:Class>
<rdf:Property rdf:about="http://www.exemple.org/garage#possede">
        <rdfs:Domain rdf:Resource="http://www.exemple.org/garage#Vehicule"/>
        <rdfs:Range rdf:Resource="http://www.exemple.org/garage#Pneus"/>
</rdf:Property>
<rdf:Property rdf:about="http://www.exemple.org/garage#fonctioneAvec">
        <rdfs:Domain rdf:Resource="http://www.exemple.org/garage#Vehicule"/>
         <rdfs:Range rdf:Resource="http://www.exemple.org/garage#Moteur"/>
</rdf:Property>
<Voiture rdf:ID="http://www.exemple.org/garage#Golf"/>
<Pneus rdf:ID="http://www.exemple.org/garage#Dunlop"/>
```

obtenir une source de connaissances qui reflète la réalité, il faut arriver à capter des subtilités additionnelles de la sémantique. Dès lors, OWL englobe le langage RDF mais avec un vocabulaire plus large et une syntaxe plus stricte [16](Lawarrée,2010). Ce dernier standard est composé de trois sous-langages que l'on peut voir dans la figure 1.11 [19] (McGuinness et van Harmelen,2003).

FIGURE 1.11 – Les trois sous-langages OWL




On a donc bien que ce ne sont pas des sous-langages disjoints mais bien des sous-ensembles. Il faut également savoir que OWL 2 possède plus que trois sous-langages.

- **OWL Lite** : OWL Lite [19] (McGuinness et van Harmelen,2003) est le sous-ensemble qui offre le moins d'expressivité, il permet de faire des hiérarchies de classes, des hiérarchies de propriétés et de définir des contraintes simples. Il répond aux besoins primaires d'une modélisation. Il existe la contrainte de cardinalité et par exemple, OWL Lite ne permet

que les cardinalités 0 ou 1. Ce sous-langage permet également de décrire qu'une propriété est l'inverse d'une autre. Par contre, cette expressivité restreinte permet de réduire la complexité de toute manipulation d'une représentation décrite en OWL Lite.

- **OWL DL** : Ceux qui utilisent OWL DL sont ceux qui veulent le plus d'expressivité mais tout en conservant le caractère calculable et décidable des conclusions [19] (McGuinness et van Harmelen,2003). L'idée c'est que l'ensemble des constructions OWL se trouve déjà dans OWL DL mais en les limitant parfois pour garder des relations correspondant à la logique descriptive, d'où le nom de ce sous-ensemble. La logique descriptive est une restriction de la logique du premier ordre avec l'idée de définir de manière formelle les concepts pertinents d'un domaine. Par rapport au langage RDFS, ce sous-ensemble rajoute essentiellement la possibilité de contraindre les cardinalités des propriétés, de déterminer si deux classes sont disjointes ou encore de décrire l'union, l'intersection ou le complément de classes. C'est de loin le sous-ensemble le plus utilisé de par son expressivité et sa calculabilité qui va permettre d'exploiter les mécanismes d'inférence de la logique descriptive.
- **OWL Full** : OWL Full est l'ensemble le plus complet, c'est celui qui exploite le plus le langage RDF et le rend totalement compatible à celui-ci. Par rapport aux deux autres sous-ensembles, il est plus expressif car il n'a aucune restriction sur le vocabulaire mais la conséquence est qu'il est également bien plus complexe. En effet, OWL Full est indécidable et si on utilise ce sous-langage, on a plus aucune garantie de calculabilité [19] (McGuinness et van Harmelen,2003).

Dans la figure 1.12, on trouve un exemple d'utilisation du langage OWL DL, où l'on raffine l'exemple utilisé dans le langage RDFS. On rajoute la notion que les classes « Voiture » et « Moto » sont disjointes [19] (McGuinness et van Harmelen,2003), qu'un véhicule peut avoir au maximum un seul moteur et on définit une classe qui représente tout ce qui n'est pas un véhicule.

### 1.1.5 Ontologie

Il est difficile de donner une définition précise de ce qu'est une ontologie. Ce qu'est une ontologie dépend du domaine dans lequel on se trouve. En philosophie, c'est une branche de la métaphysique qui s'occupe d'étudier l'être, en médecine, c'est l'étude de la genèse des maladies mais ce qui nous intéresse beaucoup plus, c'est en informatique. En informatique, une ontologie est une représentation qui est utilisée pour capturer la connaissance d'un domaine particulier, elle contient la description précise et exacte des concepts du domaine et des relations entre ceux-ci [6](Horridge,2009). Il s'agit d'une modélisation abstraite explicite et formelle du domaine. Il existe différents langages de description pour les ontologies qui apportent leurs facilités respectives mais le standard le plus récent et le plus utilisé est le standard OWL [6](Horridge,2009) présenté ci-dessus. L'ontologie a un certain intérêt dans nos objectifs car au départ ce

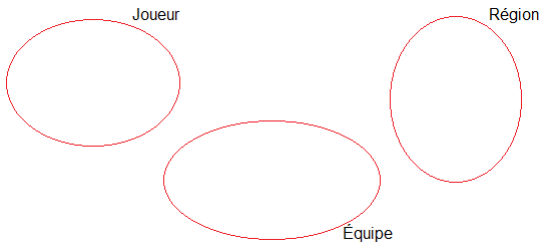
FIGURE 1.12 – Exemple d'utilisation de OWL DL

```
<owl:Class rdf:about="http://www.exemple.org/garage#Voiture">
  <rdfs:subClassOf rdf:about="http://www.exemple.org/garage#Vehicule"/>
  <owl:disjointWith rdf:Resource="http://www.exemple.org/garage#Moto"/>
</owl:Class>

<owl:Restriction>
  <owl:onProperty rdf:about="http://www.exemple.org/garage#fonctionneAvec"/>
  <owl:maxCardinality rdf:datatype="xsd:nonNegativeInteger">1</owl:maxCardinality>
</owl:Restriction>

<owl:Class>
  <owl:complementOf>
    <owl:Class rdf:about="http://www.exemple.org/garage#Véhicule"/>
  </owl:complementOf>
</owl:Class>
```

concept était destiné au web mais rien n'empêche de l'exploiter hors du web, ce que l'on a fait. Cependant, cela permet de ne pas fermer la porte à l'idée d'extraire la connaissance depuis le web et d'ensuite pouvoir l'exploiter comme nous le faisons ou inversément, de l'exploiter sur le web. De plus, une de nos contraintes principales étaient de gérer des domaines très larges et il s'avère que les ontologies permettent une représentation moins ambiguë que les systèmes experts et plus performante quand il s'agit de domaines de grande taille. Une ontologie est composée de différents composants permettant de décrire le domaine. Il y a les « Classes », les « Individus » et les « Propriétés » [6](Horridge,2009).

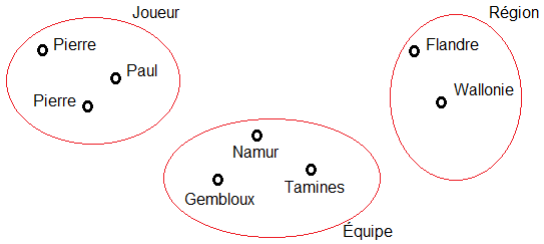
- **Classe** : Une classe [6](Horridge,2009), ou concept, est le composant le plus utilisé au sein d'une ontologie. Concrètement, il représente un groupe d'objets du domaine d'intérêt qui partagent des caractéristiques communes. Il est possible de décrire qu'un concept $A$ est un sous-concept d'un autre concept $B$. Dès lors, si un objet du domaine est « membre » du sous-concept $A$, alors il l'est également du concept $B$. Grâce à cela, on peut créer une hiérarchie de concepts et l'intérêt est que les relations de subsomption peuvent être déduites automatiquement par un moteur d'inférence [6](Horridge,2009). La figure 1.13 reprend l'exemple de trois concepts (Joueur, Région et Équipe) représentés par des classes.

FIGURE 1.13 – Exemple de classes



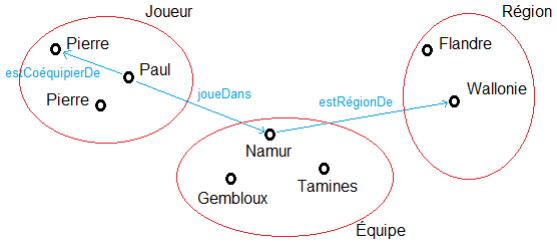
– **Individu** : Un « individu » [6](Horridge,2009) représente un objet concret de l'univers du discours qui nous intéresse. C'est l'unité de base de l'ontologie. Il peut être assimilé a une instance de classe. Quand deux « individus » représentent le même objet, alors il faut le décrire explicitement car il n'y a pas d'hypothèse de nom unique dans OWL. La figure 1.14 reprend un exemple d'instances de classes.

FIGURE 1.14 – Exemple de individuals




– **Propriété** : Les propriétés [6](Horridge,2009) servent à décrire les relations que les « individus » ont entre eux. C'est une relation qui peut être exprimée entre des « individus » ou entre des classes. Chaque propriété peut avoir sa propriété inverse. Par exemple, la propriété « contient » peut avoir comme inverse la propriété « estContenuDans ». Une propriété particulière, aussi appelée attribut [6](Horridge,2009), est une propriété qui a pour résultat une valeur effective. Une proprété peut être transitive ou symétrique. On peut trouver l'ensemble complété avec quelques propriétés dans la figure 1.15.

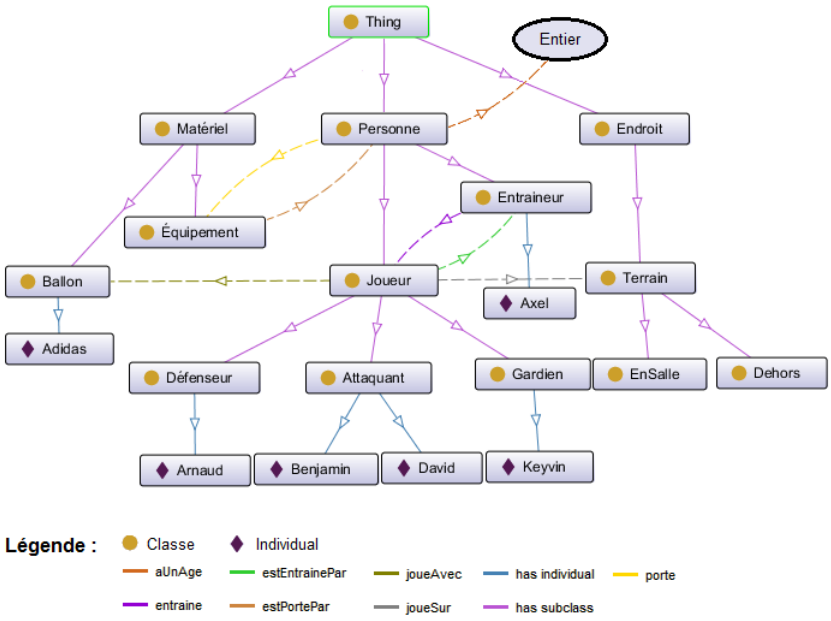
FIGURE 1.15 – Exemple de propriétés




Il y a d'autres notions qui ne font pas directement partie de la notion d'ontologie mais qui viennent la compléter pour la rendre plus facilement compréhensible et utilisable [6](Horridge,2009). Il y a la documentation : chaque classe et propriété peut être accompagnée d'une explication en langage naturel. Cette explication permet de donner toute sorte d'information comme une définition des différents concepts. Cela va permettre aux créateurs de l'ontologie de s'assurer que la représentation d'un concept dans l'ontologie respecte bien

la définition. Cela va également permettre aux utilisateurs de comprendre plus vite l'ontologie et de manière plus précise. Il y a aussi les métadonnées d'une ontologie qui vont plutôt reprendre la description générale de l'ontologie, ses auteurs, sa version, etc.

FIGURE 1.16 – Exemple de petite ontologie dans le domaine du football




Le sous-langage OWL le plus utilisé pour décrire une ontologie est le langage OWL DL car celui-ci apporte l'expressivité nécessaire tout en restant calculable. Ce langage assimilé à la logique de descriptions SHOIN(D) [1] permet d'appliquer les mécanismes d'inférence de la logique à une ontologie. Pour ce faire, il existe différents outils permettant de faire des raisonnements déterministes (Pellet [2], FaCT++ [3], TrOWL [4], HermiT [5], etc). Les deux fonctionnalités principales de l'inférence dans le domaine des ontologies sont de pouvoir déterminer si un concept est une sous-classe d'un autre concept et si une ontologie est consistante ou non, c'est à dire déterminer si toutes les classes sont instanciables. La première fonctionnalité permet de produire une hiérarchie de classes inférée [6](Horridge,2009).

1. http ://www.obitko.com/tutorials/ontologies-semantic-web/owl-dl-semantics.html
2. http ://wwww.mindswap.org/pellet
3. http ://owl.cs.manchester.ac.uk/tools/fact/
4. http ://trowl.eu/
5. http ://hermit-reasoner.com/

## 1.2 L'incertitude dans notre monde

Nous vivons dans un monde où l'incertitude prend une place importante. La majorité des événements qui se produisent sur Terre ne sont pas déterministes et ne peuvent être prédits avec précision à l'avance. Qui pourrait prédire avec exactitude ce qu'il se passera demain ? Dans la plupart des cas, personne ne le peut.

Cependant, nous pouvons parfois sentir que certains événements vont se produire. Cela peut venir d'un simple pari. Nous pouvons par exemple avoir la sensation que notre équipe de football favorite va s'imposer le weekend prochain. Mais nous pouvons également avoir des sentiments plus probables. Par exemple, il est fort probable qu'un étudiant studieux qui ne rate jamais les cours se rende à l'école en pleine semaine. Cependant, même pour des cas aussi probables, il y a des chances que cela ne se passe pas comme prévu. Dans notre exemple, l'étudiant en question pourrait tomber malade et être forcé de rester au lit toute la journée. Il pourrait très bien y avoir également d'autres alternatives comme un accident ou encore l'influence de mauvaises fréquentations qui l'inciteraient à ne pas se rendre à l'école.

Nous pouvons remarquer que l'ensemble des cas évoqués ci-dessus sont possibles. Il n'est pas possible de prédire à l'avance lequel de ceux-ci se produira. Cependant, nous pouvons sentir que certains cas sont beaucoup plus possibles que d'autres. Dans notre exemple, il est plus probable que l'étudiant se rende à l'école plutôt que de tomber malade. Nous pouvons ainsi établir un certain degré de probabilité concernant l'occurrence des événements. Cependant, bien qu'on puisse pressentir ces différents degrés de probabilité, il est difficile d'établir avec exactitude un résultat quantitatif.

Plusieurs facteurs contribuent à cette difficulté de quantifier.

Tout d'abord, il est difficile d'établir des probabilités qui seront vraies dans tous les cas. Si l'on reprend notre exemple, nous pouvons nous accorder sur le fait qu'un étudiant studieux quelconque a plus de chance de se rendre à l'école plutôt qu'être malade. Cependant, si ce même étudiant, la veille, présentait des symptômes importants d'une maladie donnée comme par exemple une toux grasse, le nez encombré ou un état fiévreux, nous pourrions cette fois accorder beaucoup plus d'importance au scénario de la maladie. Cet exemple nous montre donc qu'il y a un élément essentiel à prendre en compte lorsque nous établissons des prédictions : le contexte.

En effet, sans contexte précis, il est difficile d'établir des prédictions générales. Sans être situé géographiquement, si on nous demande quelle est la probabilité

générale qu'il pleuve demain, nous ne saurions pas vraiment répondre. Cependant, si on ajoutait à cette question un contexte géographique, il nous serait beaucoup plus facile de répondre.

En effet, si la région visée se trouve au Brésil qui subit de lourdes pluies la plupart de l'année, nous pourrions nous tourner vers une réponse positive sur le fait qu'il pleuve. Au contraire, si cette fois la région visée se trouve dans une région aride du nord de l'Afrique, la réponse d'un temps sec serait certainement plus probable.

Le contexte peut se caractériser par une multitude d'informations :

Il peut s'agir d'une situation dans le temps. Par exemple, l'année dans le cas d'un fait historique ; le mois dans le cas afin de savoir si l'événement se trouve dans la saison des pluies ; l'heure pour mesurer l'impact d'un moment de la journée sur un événement.

Une situation dans l'espace peut également devenir un élément contextuel. Il peut s'agir de la localisation d'un lieu comme celle de notre exemple mais aussi la situation d'une personne ou encore la présence d'un objet à notre proximité.

Il peut enfin s'agir de l'existence d'un événement précédent. Par exemple, si nous venons d'avoir un accident, la probabilité que l'on redouble de prudence sur la route est fortement augmentée.

A travers ces éléments de contexte, nous pouvons établir la probabilité d'occurrence d'un événement particulier. Il existe de nombreuses méthodes qui permettent de quantifier cette probabilité d'occurrence. Cependant, celles-ci ne donnent qu'une estimation du résultat. En effet, si nous voulions obtenir une probabilité exacte, il nous faudrait prendre en compte d'une infinité d'éléments contextuels reprenant également la plupart des cas improbables. C'est pour cela que les probabilités, même si elles sont quantifiées, ne restent qu'un sentiment pouvant éventuellement être appuyé par des statistiques. Si nous savons que nous avons 90% de chance de réussite, nous serons plus enclins d'effectuer une action en relation que si nous n'avions que 10% de réussite. Cependant, il nous est nécessaire de doser cette marge d'erreur (ici de 10%) et que considérer les risques que cela engendrerait.

## 1.3 Logique et incertitude

Le problème de la logique du premier ordre, on l'a vu ci-dessus, c'est qu'une formule logique qui représente une partie du domaine est soit vraie soit fausse, pas entre les deux. Or, comme exprimé précédemment, nous voulons modéliser notre monde qui est remplis d'incertitudes et de probabilités. Dès lors, de telles

formules logiques ne captent qu'une fraction de la connaissance pertinente. Les systèmes à base de faits et de règles qui reposent essentiellement sur la logique formelle du premier ordre malgré le fait qu'ils possèdent un moteur d'inférence efficace et les ontologies qui sont assimilables à la logique descriptive ont donc une applicabilité limitée en ce qui concerne les problèmes dans le domaine de l'intelligence artificielle dû à leur manque d'expressivité. Une solution à cela est de rajouter de l'information à la logique du premier ordre, de la rendre plus expressive en utilisant des modèles de représentation de la connaissance généralisés à la logique du premier ordre et qui peuvent prendre en compte la théorie des probabilités tout en conservant la capacité de s'adresser à la complexité d'un domaine.

## 1.4 Modèles graphiques probabilistes

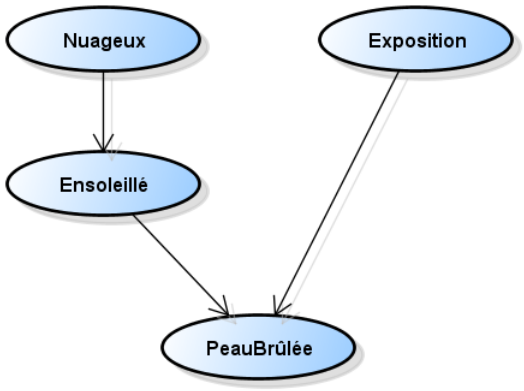
Les modèles graphiques probabilistes sont apparus spécifiquement pour combler les lacunes des différentes représentations de la connaissance. En effet, notre monde est caractérisé par sa complexité et ses événements incertains. Les modèles graphiques probabilistes combinent un modèle graphique qui répond de la théorique des graphes et des lois de probabilité qui peuvent s'adresser à la modélisation de l'incertitude [20](Bach,2006). La modularité apportée par un modèle graphique permet de décomposer un problème complexe en plus petits modules, ce qui réduit la complexité du problème de départ. Dans tous les modèles graphiques probabilistes, les nœuds modélisent des variables aléatoires et les arcs font l'hypothèse d'une dépendance conditionnelle entre les variables [20](Bach,2006). La structure sous-forme de graphe et la flexibilité d'utilisation amenée par la présence de probabilités rendent cette représentation idéale pour l'application d'algorithmes d'apprentissage et d'inférence automatiques. Le but de l'apprentissage automatique est de donner la capacité à un système d'adapter ses comportements, en se fondant sur l'analyse des données provenant du domaine alors que l'inférence a pour but que donner la capacité au système de donner automatiquement des réponses à des problèmes qui lui sont soumis et ce en prenant compte des résultats de l'apprentissage. Il existe deux grandes familles de modèles graphiques probabilistes, il y a les graphes non dirigés qu'on appelle les réseaux de Markov et les graphes dirigés non cycliques, aussi appelés réseaux Bayésiens [20](Bach,2006).

### 1.4.1 Réseau Bayésien

Dû au besoin de créer des systèmes experts qui puissent gérer des probabilités pour représenter les connaissances sont apparus les réseaux Bayésiens qui font partie des types de modèles graphiques probabilistes. Un réseau Bayésien prend la forme d'un graphe dirigé et acyclique dans lequel chaque nœud représente une variable aléatoire et chaque arc une dépendance statistique causale entre les différentes variables [21](Pérès). Le fait que le graphe soit acyclique indique qu'il ne peut y avoir de boucles dans le graphe. La figure 1.17 reprend un exemple

de réseau Bayésien.

FIGURE 1.17 – Exemple de réseau Bayésien




De plus, chaque variable est accompagnée d'un tableau de probabilités [21](Pérès). Ces probabilités peuvent soit être déterminées par des experts du domaine représenté soit par observation du domaine, ou encore, il est possible de les faire apprendre au réseau via des mécanismes d'estimation bayésienne [21](Pérès). La table de probabilités de chaque nœud représente la loi de probabilité de la variable associée à ce nœud en prenant en compte les variables qui peuvent avoir une influence sur la probabilité de la variable en question, c'est à dire les variables représentées par les nœuds parents immédiats du nœud concerné [20](Bach,2006). Par exemple, en figure 1.18, on voit la table de probabilités pour les nœuds parents « Nuageux » et « Exposition ».

FIGURE 1.18 – Tableau de probabilités

| **Tableau de probabilités** | |
|---|---|
| Il fait nuageux | 0.4 |
| Il ne fait pas nuageux | 0.6 |

| **Tableau de probabilités** | |
|---|---|
| Exposé au soleil | 0.8 |
| Non exposé au soleil | 0.2 |

Ensuite, on peut observer en figure 1.19 les probabilités des deux nœuds « Ensoleillé » et « PeauBrûlée ».

On remarque que naturellement il y a plus de chance qu'il fasse ensoleillé quand il n'y a pas de nuages ou encore que l'on a plus de chance d'avoir la peau brûlée si on est exposé à un ciel ensoleillé que si on est exposé à un ciel non ensoleillé.

Une fois que la structure du réseau est établie et qu'on a obtenu les tables de probabilités des différents nœuds, le réseau Bayésien devient une vraie machine à calculer des probabilités conditionnelles sur le domaine. Le mot « Bayésien

FIGURE 1.19 – Tableau de probabilités

| **Tableau de probabilités** | | |
|---|---|---|
| Nuageux | Vrai | Faux |
| Il fait ensoleillé | 0.2 | 0.95 |
| Il ne fait pas ensoleillé | 0.8 | 0.05 |

| **Tableau de probabilités** | | | | |
|---|---|---|---|---|
| Ensoleillé | Vrai | | Faux | |
| Exposition | Vrai | Faux | Vrai | Faux |
| Il a la peau brûlée | 0.85 | 0 | 0.25 | 0 |
| Il n'a pas la peau brûlée | 0.15 | 1 | 0.75 | 1 |

» vient du révérend Bayes qui est le créateur de la formule de probabilité conditionnelle sur laquelle est basé l'entièreté du réseau Bayésien [22](Parent et Eustache,2007). Celle-ci dit que :

$$\Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr B} = \frac{\Pr(B|A) * \Pr A}{\Pr B}$$

Si on reprend notre exemple, on a que : Pr(Ensoleillé|PeauBrûlée) = $\frac{\Pr(\text{PeauBrûlée}|\text{Ensoleillé}) * \Pr \text{Ensoleillé}}{\Pr \text{PeauBrûlée}} = \frac{0.68 * 0.65}{0.442+0.07+0+0} = 0.8633$

On a donc qu'il y a 86,33% de chance que le ciel soit ensoleillé si on a la peau brûlée.

L'intérêt de cette formule est de pouvoir faire des inductions dans les deux sens. On peut soit remonter aux causes à partir des conséquences, soit déduire les conséquences sur base des causes et ce grâce à la loi de probabilité conditionnelle et son inverse obtenue par Bayes en jouant avec la symétrie de la règle de multiplication [22](Parent et Eustache,2007). Pour être plus complet et permettre plus d'inférences, il existe deux règles qui découlent de la théorie des probabilités qui vont permettre de tout ramener à des probabilités conditionnelles [22](Parent et Eustache,2007) :

$$\Pr(A \cup B|C) = \Pr(A|C) + \Pr(B|C) - \Pr(A \cap B|C)$$

et

$$\Pr(A \cap B) = \Pr(A|B) * \Pr B = \Pr(B|A) * \Pr A$$

Plus que se baser sur la formule de Bayes, un réseau Bayésien représente la loi de probabilité conjointe des variables représentées par les nœuds du graphe dont la définition formelle est donnée par [20](Bach,2006) :

$$\Pr(X) = \prod_{x \in X} : \Pr(x|parents(x))$$

où $X$ est un ensemble de variables aléatoires contenues dans le graphe et $parents(x)$ est l'ensemble des nœuds parents immédiats du nœud représentant la variable $x$ étant donné que, sachant ses parents immédiats, chaque variable aléatoire est indépendante des nœuds non-descendants [20](Bach,2006).

Les méthodes d'inférence qui donnent des résultats précis accompagnés d'une preuve de la validité des résultats sont en générale de complexité exponentielle [22](Parent et Eustache,2007). Il existe donc des heuristiques qui permettent de réduire la complexité des méthodes mais le prix à payer est que l'on n'obtient plus des probabilités exactes mais bien des estimations de celles-ci [22](Parent et Eustache,2007). De plus, il n'y a plus aucune preuve de la validité des estimations qui est fournie.

**PR-OWL**

Dans le domaine des réseaux Bayésiens, il existe une nouvelle branche naissante qui a la volonté de rendre les ontologies capables de gérer l'incertitude car nous avons vu dans la section sur les ontologies que le langage OWL-DL avait comme limitation de ne pas pouvoir s'adresser aux probabilités. L'idée de cette branche du sémantique web est de créer une extension du langage OWL qui peut permettre de représenter des ontologies probabilistes qui se nomme PR-OWL [23](L'équipe PR-OWL/MEBN/UnBBayes). Le principe d'une extension est de conserver la compatibilité avec le langage de base tout en apportant de nouvelles définitions à celui-ci. Ce nouveau langage va permettre de faire des inférences probabilistes au sein d'un langage pour les ontologies. La sémantique de cette extension est basée sur la structure des réseaux Bayésiens à entités multiples [23](L'équipe PR-OWL/MEBN/UnBBayes).

Les réseaux Bayésiens à entités multiples(MEBN) sont des extensions aux réseaux Bayésiens qui vont permettre de représenter la notion de sous-structure répétée [23](L'équipe PR-OWL/MEBN/UnBBayes). L'idée est de modéliser la connaissance comme un ensemble de petits réseaux Bayésiens qui peuvent être instanciés et mis en relation. Le concept de MEBN a été élaboré dans le but de représenter des bases de connaissances probabilistes avec la logique du premier ordre. L'unité de base des réseaux à entités multiples est appelée « MFrag » et représente un réseau Bayésien paramétrable [23](L'équipe PR-OWL/MEBN/UnBBayes). Ces unités sont reliées par un graphe de fragments. Chaque réseau Bayésien contient la description probabiliste d'une entité du domaine et de ses relations et peut être instancié un nombre indéterminé de fois. Un « MFrag » possède une loi de probabilité conjointe locale sur l'ensemble de ses variables aléatoires. Pour obtenir la loi de probabilité conjointe de l'ensemble des « MFrags », il faut rassembler ensemble leurs lois de probabilité locale en prenant en compte les relations d'indépendance conditionnelle représentées par le graphe des « MFrags ». Un ensemble de « MFrags » à partir duquel on peut obtenir une loi de probabilité unique est appelé une « MTheory » [23](L'équipe PR-OWL/MEBN/UnBBayes). Ces « MTheories » peuvent être utilisés pour représenter des ontologies probabilistes en définissant une loi de probabilité sur les valeurs de vérité de formules de la logique du premier ordre.

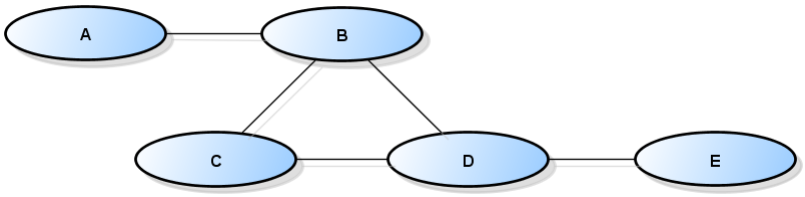
Il est difficile de parler de PR-OWL de manière plus approfondie car le langage est encore en pleine élaboration [23](L'équipe PR-OWL/MEBN/UnBBayes).

En effet, cette extension est en progression au sein de deux équipes de travail dont une s'occupe d'approfondir la sémantique du nouveau langage et l'autre s'intéresse à la construction d'un raisonneur qui s'adresse aux réseaux Bayésiens à multiple entités [23](L'équipe PR-OWL/MEBN/UnBBayes). Dès lors, cette technologie ne peut pas encore être prise en compte dans notre projet mais il est intéressant, au vu de son potentiel, de suivre son avancement pour les travaux futurs.

### 1.4.2 Réseau de Markov

L'idée de base est que la connaissance des distributions de probabilité de chacune des variables ne suffit pas. Tout comme les réseaux Bayésiens, les réseaux de Markov sont au sens abstrait des modèles pour la distribution conjointe d'un ensemble de variables aléatoires. La notion de distribution conjointe va permettre de prendre en compte la covariance entre les différentes variables. Plus concrètement, un réseau de Markov est représenté par un graphe non dirigé dans lequel chaque nœud représente une variable et chaque arc une dépendance statistique entre deux variables [4](Richardson et Domingos,2006). Sachant ses voisins, chaque variable est indépendante du reste du graphe. Cette représentation est similaire à un réseau Bayésien à deux différences près : un réseau Bayésien est dirigé et acyclique [20](Bach,2006). La figure 1.20 montre un exemple de réseau de Markov.

FIGURE 1.20 – Exemple de réseau de Markov



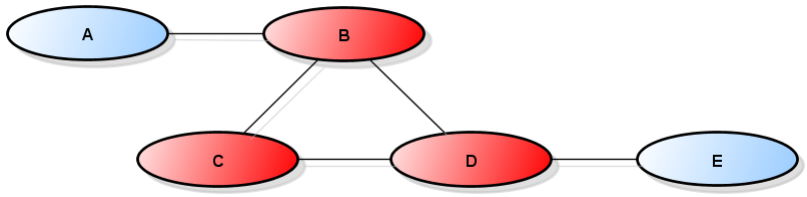

Une clique d'un graphe non orienté, voir figure 1.21, est un sous-ensemble de sommets de ce graphe dont le sous-graphe induit est complet, c'est à dire que tous sommets de la clique pris deux à deux possèdent un arc les reliant. Dans un réseau de Markov, il faut que le graphe soit non dirigé, avoir un ensemble de fonctions de compatibilité $\phi_c$ pour chaque clique du graphe [4](Richardson et Domingos,2006). Une fonction de compatibilité lie un nombre réel non négatif selon les valeurs des membres d'une clique.

La distribution conjointe représentée par le modèle de Markov est donnée par [4](Richardson et Domingos,2006) :

$$\Pr(X = x) = \frac{1}{Z} \prod_{c \in C} \phi_c(x_c)$$

où $x_c$ contient les valeurs de vérités des variables aléatoires se trouvant dans la c$^{\text{ème}}$ clique du graphe et $Z$ est la fonction dite de partition qui, on le verra

FIGURE 1.21 – L'ensemble des noeuds rouges représente une clique du réseau




par la suite, est source d'un certain nombre de contraintes de complexité. Cette fonction de partition qui est constante permet de normaliser le produit et obtenir une probabilité cohérente [4](Richardson et Domingos,2006). Donc, la fonction de partition $Z$, que l'on ne retrouve pas dans la distribution conjointe des réseaux Bayésiens, est donnée par [4](Richardson et Domingos,2006) :

$$Z = \sum_{x \in X} \prod_{c \in C} \phi_c(x_c)$$

Si on considère par exemple la fonction de compatibilité suivante :

$$\phi(A,B) = \begin{cases} 2.8, & \text{si } A \wedge \neg B \\ 2.0, & \text{si } A \wedge B \\ 1.5, & \text{sinon} \end{cases}$$

$$\phi(B,C,D) = \begin{cases} 2.7, & \text{si } \neg B \wedge \neg C \wedge D \\ 0.4, & \text{si } B \wedge \neg C \wedge \neg D \\ 6.0, & \text{sinon} \end{cases}$$

$$\phi(D,E) = \begin{cases} 4.6, & \text{si } D \wedge E \\ 2.5, & \text{si } D \wedge \neg E \\ 0.7, & \text{sinon} \end{cases}$$

En s'épargnant les détails du calcul car il y a 25 combinaisons de variables possibles, on a que $Z$ = 667,411 et dès lors on a que $\Pr(A,B,C,D,E)$ = 55,2/667,411 = 0,082708 ou encore que $\Pr(A,B,\neg C,\neg D,\neg E)$ = 77,28/667,411 = 0,115791

Toutefois, il est conventionnel de remplacer la fonction de compatibilité par une fonction exponentielle de la somme des fonctionnalités de chaque état [4](Richardson et Domingos,2006). De plus, à chaque fonctionnalité est associée un poids. Il est courant de limiter cette fonctionnalité à un résultat binaire. On a [4](Richardson et Domingos,2006) :

$$\Pr(X = x) = \tfrac{1}{Z} \exp\left(\sum_j w_j f_j(x)\right)$$

et :

$$Z = \sum_{x \in X} \exp\left(\sum_j w_j f_j(x)\right)$$

Il existe une fonctionnalité correspondant à chaque état de chaque clique et, dès lors, la taille de la représentation est exponentielle selon le nombre de cliques dans le graphe. Le poids de chaque fonctionnalité est logiquement déterminé par la formule : $\log \phi_c(x_c)$ [4](Richardson et Domingos,2006). Mais l'avantage de cette représentation plus conventionnelle est que l'on peut élaborer un nombre plus restreint de fonctionnalités, ce qui permet d'obtenir une représentation plus abordable que la première forme de représentation avec la fonction de compatibilité.

Pour représenter un domaine de même taille, le graphe contenu dans un réseau Bayésien peut devenir significativement plus large que celui contenu dans un réseau de Markov [24](Srihari), ce qui peut poser des problèmes de mise à l'échelle lorsqu'on traite des domaines assez grands. La notion de dépendance dans un réseau Bayésien est nettement plus compliquée car elle prend en compte le caractère directionnel des arcs dans le graphe. Bien qu'un réseau Bayésien montre de meilleures performances au niveau de l'apprentissage automatique [24](Srihari), nous avons choisi de continuer notre travail avec les réseaux de Markov car ils sont plus aptes à gérer les ensembles de données très larges et c'est l'intérêt premier de notre outil.

### 1.4.3 Réseau logique de Markov

Nous avons montré ci-dessus que le monde réel dans lequel on vit est à la fois complexe et rempli d'incertitude. Le problème est qu'il s'agit de domaines appartenant à ce monde que l'on veut traiter. Dès lors, pour pouvoir raisonner, il va falloir prendre ces deux aspects en compte. La logique du premier ordre permet de traiter la complexité alors que la théorie des probabilités permet de s'adresser à l'incertitude. Les réseaux logiques de Markov (MLN) vont permettre de combiner les deux en appliquant un modèle de Markov à la logique du premier ordre.

Un réseau logique de Markov est un ensemble de formules de la logique du premier ordre ainsi qu'un poids associé à chaque formule [4](Richardson et Domingos,2006). Un poids est toujours un nombre réel. Cependant, il faut faire attention qu'il peut exister certaines restrictions selon l'outil qui implémente le réseau de Markov. Certaines implémentations ne permettent pas d'exprimer l'entièreté de la syntaxe de la logique et d'autres n'acceptent pas les poids négatifs. On remarque donc que la notion de réseaux logiques de Markov fait référence à une représentation de l'information qui va permettre de construire une instance d'un modèle de Markov particulier où les nœuds du graphe représentent les différents atomes et les arcs représentent les connecteurs logiques qui lient les atomes pour construire une formule logique [4](Richardson et Domingos,2006). Chaque formule est une clique dans le graphe. La fonctionnalité de chaque clique qui remplace la fonction de compatibilité est tout simplement la fonction qui est vraie quand la formule logique représentée par la clique a pour valeur de vérité

vrai et fausse sinon [4](Richardson et Domingos,2006). La figure 1.22 montre un exemple de transformation en réseau logique de Markov.

FIGURE 1.22 – Exemple de transformation en réseau logique de Markov

| **Français** | **Logique du premier ordre** | **Forme normale conjonctive** | **Poids** |
|---|---|---|---|
| Si x est le descendant de y et que y est le descendant de z alors x est le descendant de z | $\forall x \forall y \forall z$ Descendant(x,y)$\wedge$ Descendant(y,z) $\Rightarrow$ Descendant(x,z) | $\neg$Descendant(x,y)$\vee$ $\neg$Descendant(y,z)$\vee$ Descendant(x,z) | 5.5 |
| Le stress accentue le fait d'être cardiaque | $\forall x$ Stress(x) $\Rightarrow$ Cardiaque(x) | $\neg$Stress(x) $\vee$ Cardiaque(x) | 1.5 |
| Si x est le descendant de y est que y est stressé alors x est stressé | $\forall x \forall y$ Descendant(x,y) $\wedge$ Stress(y) $\Rightarrow$ Stress(x) | $\neg$Descendant(x,y)$\vee$ $\neg$Stress(y) $\vee$ Stress(x) | 0.8 |

Le problème est qu'une formule atomique (exemple : $Stress(x)$) n'a pas encore de valeur de vérité. Il faut donc avant tout instancier la formule logique de départ avec un ensemble de constantes. L'algorithme d'instanciation est assez simple mais l'on verra par la suite qu'il sera possible de l'améliorer en permettant de ne pas instancier l'entièreté du réseau pour des raisons de performance lors de l'inférence qui va suivre [4](Richardson et Domingos,2006). Avant d'appliquer l'instanciation, il faut transformer la formule de départ en forme normale conjonctive en respectant la méthode vue dans la section sur la logique du premier ordre, mis à part la skolémisation. Ici, les formules quantifiées existentiellement sont remplacées par des disjonctions de leurs instanciations sur $C$. L'algorithme 1 montre le pseudo code du procédé générale d'instanciation [4](Richardson et Domingos,2006).

On remarque tout de suite que le procédé est scindé en deux étapes. Dans la première, on prend les clauses de la formule une par une et on instancie chaque variable de chaque clause avec toutes les constantes. Dès lors, le nombre de constantes et surtout le nombre de variables différentes dans les clauses influencent le nombre de clauses finales instanciées. On a que le nombre de clauses instanciés d'une formule vaut [25](Niu, Ré, Doan et Shavlik,2010) :

$$\sum_i \left( n_i \cdot |C|^i \right)$$

où $n_i$ est le nombre de clauses de la formule contenant $i$ variable(s) différente(s). L'union ensembliste présente pour ajouter les nouvelles clauses instanciées aux existantes sert à ne pas avoir deux clauses identiques. En second lieu, on part des clauses instanciées obtenues à la première étape et on réitère sur chacune

**Algorithm 1** Instanciation

**Require:** $F$ est une formule logique du premier ordre et $C$ est un ensemble de constantes

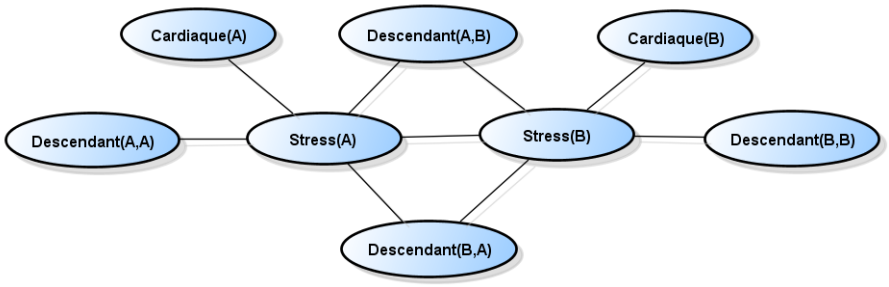
**Ensure:** $G_F$ contient l'ensemble des clauses instanciées

$G_F = \emptyset$
**for all** clause $F_j \in F$ **do**
  $G_j = \{F_j\}$
  **for all** variable $x$ contenue dans $F_j$ **do**
    **for all** clause $F_k(x) \in G_j$ **do**
      $G_j \Leftarrow G_j \backslash F_k(x) \cup \{F_k(c_1), ..., F_k(c_{|C|})\}$ , où $F_k(c_i)$ est $F_k(x)$ avec $x$ remplacé par $c_i \in C$
      $G_F \Leftarrow G_F \cup G_j$
    **end for**
  **end for**
**end for**
**for all** clause instanciée $F_j \in G_F$ **do**
  **repeat**
    **for all** fonction $f(a_1, a_2, ...)$ dont tous les arguments sont des constantes **do**
      $F_j \Leftarrow F_j$ avec $f(a_1, a_2, ...)$ remplacé par la constante $c$ où $c = f(a_1, a_2, ...)$
    **end for**
  **until** $F_j$ ne contienne plus de fonctions
**end for**

d'elles le processus de substitution de fonctions jusqu'à ce qu'il n'existe plus de fonctions. Ce processus consiste à sélectionner l'ensemble des fonctions dont les arguments sont des constantes et à les remplacer par la constante correspondante. La constante a donc la valeur de la fonction qu'elle substitue.

Supposons l'ensemble de constantes : $Alex(A), Bernard(B)$ et les deux dernières règles de l'exemple ci-dessus, cela produira un réseau de Markov instancié en figure 1.23.

FIGURE 1.23 – Exemple de réseau de Markov instancié




Le nombre de constantes peut faire varier de manière significative la taille du réseau de Markov. Toutes les formules résultantes d'une instanciation auront le même poids que la formule logique de laquelle elles découlent [4](Richardson et Domingos,2006). Vous pouvez trouver en annexe A.1.2 un exemple plus large d'instanciation de réseau logique de Markov. Une fois que l'on a ce réseau instancié, on peut l'utiliser pour déduire des probabilités concernant le domaine d'application. Par exemple, on peut s'interroger sur la probabilité que Bernard soit cardiaque si Alex est stressé et que Bernard est un descendant de Alex. Pour ce faire, voici la loi de distribution de réseau de Markov instancié [4](Richardson et Domingos,2006) :

$$\Pr(X = x) = \frac{1}{Z} \exp\left(\sum_i w_i n_i(x)\right)$$

où $n_i$ est le nombre d'instanciations vraies de la i$^{\text{ème}}$ formule de la logique du premier ordre de départ dans l'état $x$ des constantes. Nous reviendrons un peu plus sur le mécanisme d'inférence dans une section suivante. Après explications, voici la définition purement théorique d'un réseau logique de Markov [4](Richardson et Domingos,2006) :

**DEFINITION** Un réseau logique de Markov $L$ est un ensemble de couples $F_i, w_i)$, où $F_i$ est une formule de la logique du premier ordre et $w_i$ est un nombre réel. Étant donné un ensemble fini de constantes $C$, cela définit un réseau de Markov $M_{L,C}$ comme suit :

1. $M_{L,C}$ contient un nœud binaire pour chaque instanciation possible de chaque prédicat apparaissant dans $L$. La valeur du nœud est 1 si l'atome instancié est vrai, et 0 sinon.

2. $M_{L,C}$ contient une fonctionnalité pour chaque instanciation possible de chaque formule $F_i$ dans $L$. La valeur de cette fonctionnalité est 1 si la formule instanciée est vraie, et 0 sinon. Le poids de la fonctionnalité est le poids $w_i$ associé à $F_i$ dans $L$.

### Apprentissage de poids

**Génératif** On a vu qu'un réseau logique de Markov est constitué d'un ensemble de formules de la logique du premier ordre que l'on obtient en modélisant la connaissance que l'on a du domaine qui nous intéresse, d'un ensemble de poids associés à chaque formule et d'un ensemble de constantes qui représentent l'ensemble des objets du domaine. On remarque que la seule chose que l'on ne connaît pas à cette étape-ci, ce sont les différents poids des formules. Les poids sont très importants car on voit bien quand on regarde la loi de distribution que ceux-ci vont influencer de manière exponentielle la probabilité d'avoir un certain monde possible plutôt qu'un autre. Cependant, nous n'avons pas assez d'informations pour pouvoir les calculer. En effet, il faut un ensemble d'évidences pour pouvoir entraîner le modèle de Markov et calculer les poids [4](Richardson et Domingos,2006). Une évidence est une prédicat instancié qui a pour valeur de vérité vrai. Sur cet ensemble, on fait l'hypothèse qu'on est dans un monde fermé, c'est à dire que tous les prédicats instanciés possibles qui ne se trouvent pas dans l'ensemble d'entraînement sont considérés comme ayant pour valeur de vérité faux [4](Richardson et Domingos,2006). Une fois que l'on a cet ensemble, l'apprentissage de poids se résume à un problème d'optimisation convexe [4](Richardson et Domingos,2006). Ce qui veut dire que le critère que l'on essaie de minimiser est convexe. Il existe un autre type d'apprentissage que l'on peut appliquer dans le domaine des réseaux logiques de Markov, c'est l'apprentissage de la structure [26](Dinh,2011). A partir d'un ensemble de données d'apprentissage, souvent elles contiennent la description du domaine mais sans les relations entre les différentes données, on peut selon différentes techniques générer des clauses pondérées en vue de créer la structure d'un futur réseau logique de Markov. Cependant, nous allons uniquement nous focaliser sur l'apprentissage de poids car nous possédons déjà la structure du réseau logique de Markov que l'on extrait de l'ontologie. Nous détaillons plus en détails le passage d'un ontologie à un réseau de Markov dans la partie développement.

Un problème d'optimisation [27](Cohen) est un problème qui consiste à trouver, au sein d' un ensemble d'alternatives qui satisfont une propriété, l'élément qui optimise une certaine fonction de coût. Cette dernière peut également être appelée « objectif » ou encore « critère ». On peut soit essayer de minimiser la fonction ce qui revient à un problème de minimisation soit de maximiser la fonction et là on a un problème de maximisation. Si on considère $X$ l'ensemble des alternatives et $f : X \to R$ la fonction de coût, on cherche à trouver $x' \in X$ tel que :

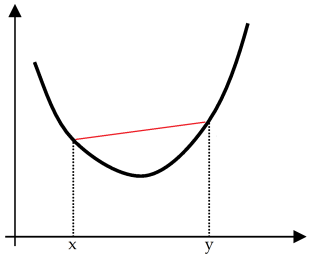
$$\forall x \in X, x \neq x' : f(x') < f(x) \text{ (problème de minimisation)}$$

ou

$$\forall x \in X, x \neq x' : f(x) < f(x') \text{ (problème de maximisation)}$$

Une des sous-disciplines de l'optimisation est l'optimisation convexe. Dans les problèmes d'optimisation convexe, qui sont plus simples à résoudre, la fonction de coût et l'ensemble des alternatives sont convexes. Dans le cas unidimensionnel, une fonction est dite convexe [27](Cohen), figure 1.24, si la valeur de la fonction $f$ en tout point du segment $[x, y]$ est situé sous la valeur de la ligne du segment en ce même point.

FIGURE 1.24 – Exemple de fonction convexe

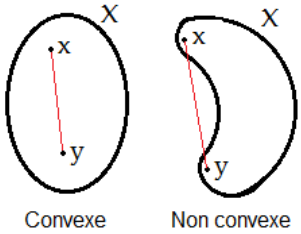



Soit $E$ un espace vectoriel sur $R$. On appelle segment de $E$ de $x$ à $y$, l'ensemble définit comme suit [28](Gilbert,2007) :

$$[x, y] = \{(1 - z) * x + z * y | z \in [0, 1]\} \text{ , où} x \text{et} y \in E.$$

Une partie $X$ de $E$ est dite convexe [28](Gilbert,2007), figure 1.25, si $\forall x, y \in X$, le segment $[x, y]$ est contenu dans $X$.

FIGURE 1.25 – Exemple d'ensembles convexes et non convexes




Dans le cas de l'apprentissage de poids, l'ensemble des alternatives est l'ensemble des vecteurs contenant les poids et la fonction de coût est la fonction de vraisemblance à maximiser.

Ici, le problème revient à trouver les poids qui maximisent le produit de leur probabilité et la vraisemblance des données [4](Richardson et Domingos,2006).

On parlait d'un problème de minimisation ci-dessus, on va donc plutôt minimiser la log-vraisemblance négative, ce qui est équivalent. On dérive la log-vraisemblance comme ceci [4](Richardson et Domingos,2006) :

$$n_i(x) - \sum_{x' \in X} \Pr_w(X = x') n_i(x')$$

où $x$ représente un état possible de la base de données des évidences alors que $x'$ représente tous les états possibles de la base de données et $n_i$ est le nombre d'instanciation vraie en considérant la i$^{\text{ème}}$ formule. On a que $\Pr_w(X = x')$ se calcule comme $\Pr(X = x')$ avec l'ensemble de poids courants $w$. Cela revient à calculer la différence entre le nombre d'instanciations vraies de la formule et la prévision en accord avec le modèle courant [4](Richardson et Domingos,2006). Ce calcul pose deux problèmes au niveau de la traitabilité. En effet, compter le nombre d'instanciations vraies d'une formule et le nombre d'instanciations vraies attendues est intraitable de manière efficace et donc le gradient utilisé dans la méthode d'optimisation se basant sur la log-vraisemblance ne converge pas dans des temps raisonnables [4](Richardson et Domingos,2006). Pour traiter ce problème, il est courant dans de nombreux domaines de recherche de remplacer la log-vraisemblance par une pseudo-vraisemblance dans la méthode d'optimisation [4](Richardson et Domingos,2006) :

$$\Pr^*_w(X = x) = \prod_{l=1}^{n} \Pr_w(X_l = x_l | MB_x(X_l))$$

où $MB_x(X_l)$ est l'état des variables de la couverture de Markov de $X_l$. Dans un réseau de Markov, la couverture de Markov d'un nœud est l'ensemble de ses nœuds voisins dans le réseau. La pseudo-vraisemblance revient à calculer le produit des vraisemblances conditionnelles de chaque variable étant donné la valeur de ses voisins dans le réseau [4](Richardson et Domingos,2006). On peut calculer cette expression sans utiliser d'inférence sur le modèle et donc, dans des temps plus raisonnables que la vraisemblance présentée au départ.

**Discriminatif** Jusqu'à présent nous parlions de l'apprentissage de poids génératif mais il existe un autre type d'apprentissage de poids qui est discriminatif. En effet, parfois lors de l'apprentissage, on connaît déjà la requête que l'on va soumettre au réseau. On peut donc la prendre en compte durant le processus d'apprentissage [29](Singla et Domingos). Pour ce faire, il faut redéfinir la loi de probabilité conditionnelle pour prendre en compte la requête. Si on considère un réseau logique de Markov avec $X$ : l'ensemble des atomes formant la requête et $Y$ : l'ensemble des évidences, on a [29](Singla et Domingos) :

$$\Pr(x = X | y = Y) = \frac{1}{Z} \exp\left(\sum_{i \in G_x} w_i g_i(y, x)\right)$$

où on ne considère plus que les clauses instanciées qui impliquent un atome se trouvant dans la requête ($G_x$) et la fonction $g_i(x, y)$ vaut 1 si la i$^{\text{ème}}$ clause instanciée est vraie et 0 sinon.

Les différents algorithmes qui existent pour implémenter cet apprentissage de poids utilisent des heuristiques pour approximer les parties de la formule qui sont non calculables [30](Lowd et Domingos). Parmi ces divers algorithmes il s'avère que celui qui montre les meilleurs résultats est l'algorithme de gradient conjugué avec préconditionnement [30](Lowd et Domingos).

**Inférence**

Une fois que l'on a la structure du graphe instancié et qu'on lui a fait apprendre l'ensemble des poids associés aux différentes clauses représentées par le graphe, on peut utiliser le réseau logique de Markov et l'ensemble d'évidences pour inférer des résultats. L'inférence permet principalement de déduire deux types de conclusions [25](Niu et al.,2010). Tout d'abord elle permet de déterminer l'état du monde possible le plus probable, on l'appelle l'inférence du maximum a posteriori(MAP) car elle permet de connaître le monde avec la vraisemblance maximale [25](Niu et al.,2010). Elle permet également de calculer des probabilités conditionnelles comme l'on a vu au sein des réseaux Bayésiens, on l'appelle l'inférence marginale [25](Niu et al.,2010). Dans le cadre de nos objectifs d'aide à la décision, nous allons nous intéresser exclusivement aux déductions de probabilités conditionnelles.

Si on considère un réseau logique de Markov après apprentissage des poids, un ensemble de constantes, un ensemble d'évidences et $F_1$, $F_2$ des formules de la logique du premier ordre, on a selon le théorème de Bayes que [4](Richardson et Domingos,2006) :

$$\Pr(F_1|F_2) = \frac{\Pr(F_1 \wedge F_2)}{\Pr(F_2)} = \frac{\sum_{x \in \mathcal{X}_{F_1} \cap \mathcal{X}_{F_2}} \Pr(X=x)}{\sum_{x \in \mathcal{X}_{F_2}} \Pr(X=x)}$$

où $\mathcal{X}_{F_i}$ est l'ensemble des mondes dans lesquels $F_i$ est vraie et $\Pr(x)$ est la loi de probabilité définie pour les réseaux logiques de Markov. Si on reprend notre petit exemple, en essayant de répondre à la question « quelle est la probabilité que Bernard soit cardiaque si Alex est stressé et que Bernard est un descendant d'Alex ? », on a que $F_1 = Cardiaque(B)$, $F_2 = Stress(A) \wedge Descendant(B, A)$ et :

$$\Pr(Cardiaque(B)|Stress(A), Descendant(B, A)) = \frac{\Pr(Cardiaque(B), Stress(A), Descendant(B,A))}{\Pr(Stress(A), Descendant(B,A)}$$

Le processus d'inférence dans les réseaux logiques de Markov englobe deux autres types d'inférence : l'inférence probabiliste et l'inférence logique [4](Richardson et Domingos,2006). Malgré le fait que l'inférence au sein des réseaux logiques de Markov est souvent plus rapide que dans les autres modèles graphiques probabilistes, ces deux types d'inférence restent intraitables en des temps raisonnables [4](Richardson et Domingos,2006). Comme expliqué dans la section sur les réseaux Bayésiens, il existe des heuristiques pour le mécanisme

d'inférence qui réduiront la complexité en faisant une approximation du résultat [4](Richardson et Domingos,2006). Cette approximation est obtenue en appliquant des méthodes d'échantillonnage. Pour les méthodes qui vont suivre, on fait l'hypothèse que la requête et les évidences soient de la forme d'une conjonction d'atomes instanciés.

Les méthodes d'échantillonnage peuvent produire des résultats différents à chaque exécution et sont des alternatives aux méthodes déterministes. Elles reposent sur un mécanisme de sélection aléatoire et doivent s'assurer de définir une distribution qui permet à tous les éléments sélectionnables d'avoir une chance égale d'être choisis. Une fois que l'on a la requête, l'objectif de ces méthodes est de produire à chaque étape un échantillon de la probabilité de la requête [4](Richardson et Domingos,2006). Au final, une fois que la séquence d'échantillons converge, l'ensemble des échantillons qui satisfont la requête représentera une approximation de la probabilité recherchée [4](Richardson et Domingos,2006). Un des types de méthode d'échantillonnage le plus répandu est le « Monte Carlo Markov Chain » (MCMC) [31](Kroc). On l'appelle ainsi parce que c'est une méthode qui utilise uniquement la valeur de l'échantillon précédent pour générer l'échantillon suivant et ainsi former une chaîne de Markov [31](Kroc). Une chaîne de Markov est une chaîne qui respecte la propriété de Markov qui veut qu'aucun élément supplémentaire à l'état présent ne soit nécessaire pour prédire l'état futur [31](Kroc). Il existe diverses méthodes d'échantillonnage de type MCMC [32](Poon et Domingos) mais il a été prouvé de manière expérimentale que l'algorithme MC-SAT qui effectue des échantillonnages par tranche et teste la satisfiabilité de ceux-ci montre de meilleures performances que l'échantillonnage de Gibbs ou encore l'échantillonnage par « simulated tempering » [32](Poon et Domingos).

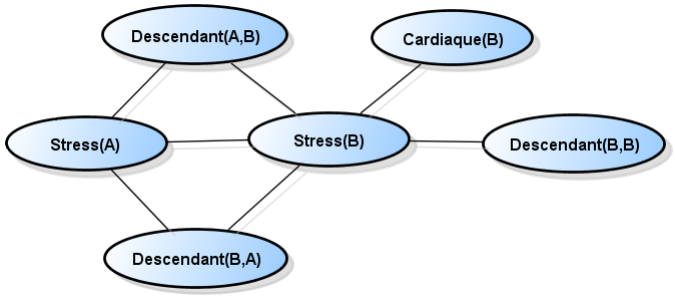
En plus de ces méthodes d'approximation, il est possible de réduire la complexité du mécanisme d'inférence en réduisant la taille du réseau de Markov instancié [4](Richardson et Domingos,2006). Plus le réseau est petit et plus le processus d'inférence est accéléré. L'algorithme 2 permet d'obtenir le réseau le plus petit possible pour pouvoir calculer la probabilité, en fonction de la requête et des évidences [4](Richardson et Domingos,2006).

Dans l'aglorithme 2, $MB(q)$ est la couverture de Markov du nœud représentant l'atome instancié $q$. L'idée de cet algorithme est assez simple, on démarre de l'ensemble des atomes instanciés qui constituent la requête et on itère tant que cette ensemble n'est pas vide. A chaque itération on rajoute à l'ensemble de départ les nœuds de la couverture de Markov de chaque élément mise à part les atomes que l'on retrouve dans l'ensemble des évidences et c'est cela qui permet de réduire le réseau de départ. Si on reprend notre exemple de réseau de Markov instancié on aura la figure 1.26 après réduction.

**Algorithm 2** Réduction de réseau de Markov

**Require:** $F_1$, l'ensemble formant la requête
$F_2$, l'ensemble des évidences
Un réseau logique de Markov instancié
**Ensure:** $G$ contenant l'ensemble des noeuds du réseau réduit en gardant les arcs entre les noeuds présents dans le réseau logique de départ
$G \Leftarrow F_1$
**while** $F_1 \neq$ **do**
  **for all** atome instancié $q \in F_1$ **do**
    **if** $q \notin F_2$ **then**
      $F_1 \Leftarrow F_1 \cup (MB(q) \setminus G)$
      $G \Leftarrow G \cup MB(q)$
    **end if**
    $F_1 \Leftarrow F_1 \setminus \{q\}$
  **end for**
**end while**

FIGURE 1.26 – Exemple de réseau de Markov instancié réduit



# Chapitre 2

# Objectifs

**Sommaire**



## 2.1 But final

Le domaine médical est un domaine à la fois large et très compliqué. La précision est indispensable lors d'analyses ou de traitements et la moindre petite erreur peu mener à des conséquences désastreuses. Un des problèmes de ce domaine est l'étendue de sa matière. La médecine n'est pas une science simple mais une multitude de sciences compliquées (orthopédie, cardiologie, endocrinologie, ...). De par son ampleur, il est impossible qu'un médecin, seul, puisse s'y connaître dans tous les domaines qu'il est amené à traiter. De ce fait, un diagnostic peut être divisé entre plusieurs professionnels de la médecine, chacun traitant la partie qui lui est propre. Cette approche pluridisciplinaire nécessite par ailleurs la collaboration des différents experts, ce qui peut également mener à des mauvaises décisions suite à un manque de communication ou une incompréhension entre les différents intervenants.

De plus, un autre problème peut intervenir pour les patients se rendant à l'hôpital pour des problèmes peu spécifiques. Le diagnostic, réalisé par un médecin généraliste, pourrait s'avérer compliqué et l'expert généraliste pourrait ne pas savoir vers lequel de ses collègues se diriger. En effet, le nombre de symptômes possibles étant à la fois important, mais aussi combinatoire, les conclusions peuvent être relativement difficile à tirer dans certains cas particuliers. Sans compter le fait que la plupart des symptômes, seuls, ne signifient pas grand-chose. Pour tirer une conclusion efficace, il faut analyser l'ensemble de

ceux-ci comme un tout et déduire de quel type de maladie le patient semble être atteint.

Intuitivement, nous pourrions déjà constater que ce raisonnement semble induire des probabilités et des règles. En effet, un symptôme seul étant attaché à une maladie pourrait être vu comme une règle. Un exemple pourrait être le fait qu'un patient tousse énormément lorsqu'il va voir le médecin. Ce dernier pourrait supposer que son patient est atteint d'une maladie de la gorge comme une bronchite ou une pharyngite. La règle induite serait donc « Tousser implique une maladie de la gorge ». Maintenant, nous pourrions pousser le raisonnement plus loin. Il nous faudrait cette fois intégrer le contexte dans lequel se trouve le patient traité, ainsi que l'ensemble de ses symptômes comme un tout. Ainsi, si nous reprenons l'exemple précédent, le médecin pourrait cette fois-ci considérer le contexte du malade qui a fumé pendant ces trente dernières années. Il pourrait également considérer d'autres symptômes qui seraient une douleur thoracique lorsque son patient respire, une voix enrouée et une perte de poids. Le tout mis ensemble, il pourrait maintenant redouter un cancer du poumon et le diriger vers le service approprié.

Dans ce cas-ci, l'intuition du docteur d'amener son patient à réaliser des examens plus approfondis concernant ce type de cancer pourrait être vu comme une probabilité. Si l'on demande à cet expert quel est son degré de certitude, il ne pourra certainement pas répondre avec précision. Cependant, l'idéal serait, à partir de symptômes, d'un contexte et de règles établies sur ceux ci, d'arriver à en dériver une probabilité précise concernant les risques d'un patient d'avoir telle ou telle maladie.

C'est dans cette optique que nous travaillons afin de mettre en place un outil d'aide à la décision permettant aux médecins d'obtenir un ensemble de maladies possibles auxquelles sont attachées des probabilités à partir de règles provenant des symptômes et du contexte dans lequel se trouve un patient donné.

Notre travail intégrera le projet ORTHOGEN qui travaille en collaboration avec l'hôpital de Mont-Godinne. Il s'agit d'un projet dont la finalité est de fournir un programme d'aide à la décision dans le domaine médical [2](Schobbens et al., 2013). L'outil qui sera proposé ressemblera à la description faite ci-dessus. Cependant, pour les raisons d'ampleur de la médecine en général, il ne traitera que d'une sous partie de celui-ci. En l'occurrence, il s'agira des diagnostics concernant les infections orthopédiques. Ce domaine bien particulier est à la fois critique et compliqué. Il est très important que les analyses réalisées ne soient pas biaisées sans quoi des infections pourraient survenir pour les personnes ayant reçu une prothèse orthopédique. Ces infections, suivant le cas, peuvent engendrer des conséquences plus ou moins graves. La complexité de ce domaine provient du fait que ces infections ne peuvent se déclarer et se développer que plusieurs mois après la pose de la prothèse. Il n'est donc pas toujours facile de déceler

les symptômes pouvant conduire à celles-ci, bien qu'une réaction rapide soit nécessaire lorsque celles-ci surviennent.

Pour réaliser sa tâche, le projet ORTHOGEN a mis en place, grâce à la consultation d'un grand nombre de spécialistes, une ontologie reprenant tout le domaine d'application qu'il devra traiter [3](Schobbens et al., 2013). Cette ontologie comporte l'ensemble des règles liant un symptôme ou un élément contextuel à une possible infection ou autre problème. C'est cette connaissance qui sera utilisée afin de déduire dans quel cas pourrait se trouver un patient se plaignant de symptômes suite à une opération orthopédique.

Afin de raisonner efficacement sur l'ensemble de règles qui compose l'ontologie, ORTHOGEN va travailler conjointement avec un moteur d'inférence probabiliste basé sur la logique de Markov. C'est ce processus de déduction qui va amener la notion d'incertitude et ainsi induire les probabilités que nous avons évoquées précédemment. C'est par ailleurs en tenant compte de l'incertitude de certaines connaissances que le moteur d'inférence probabiliste va tenter de modéliser un raisonnement efficace qui devra être le plus proche de la réalité.

Nous voyons déjà ici que le choix de ce moteur d'inférence probabiliste est crucial dans le processus de calcul. En effet, la nécessité d'obtenir un degré de précision important oblige le projet ORTHOGEN à choisir un outil aussi fiable qu'efficace pour effectuer ses raisonnements probabilistes. Il s'agit d'ailleurs d'un enjeu très important, car le programme d'aide à la décision qui devra être le résultat de ce projet ne peut se permettre d'être imprécis. Induire en erreur un expert médical pourrait l'amener à réaliser les mauvais choix et par conséquent tirer les mauvaises conclusions pour un patient. Cette simple erreur informatique pourrait alors mettre en danger des vies humaines. Nous voyons donc par cet exemple qu'un des enjeux principaux qui ne doit pas être oublié par ce projet est le risque d'atteinte aux vies humaines concernées en cas d'erreur ou d'imprécision trop importante du programme.

De plus, une autre limite à considérer serait la substitution du raisonnement, anciennement fait par un médecin, par une machine. Le projet vise à rassembler l'ensemble de la connaissance possédée par les experts médicaux dans un logiciel permettant de tirer des conclusions à la place de ce dernier. Bien sûr cet outil ne cherche pas à substituer le médecin par un ordinateur. Au contraire, le but est de l'aider dans la prise de sa décision en lui fournissant l'ensemble des cas possibles. Cela lui permet d'une part d'avoir un premier aperçu sur le type de problème qui semble se dégager du diagnostic et, d'autre part, cela lui permet également de ne pas omettre des cas possibles. Mais la question qu'il ne faut pas oublier est l'impact de ce projet sur l'expert médical. Celui-ci est pour le moment obligé de retenir tous les cas possibles au quotidien. Si d'ici une dizaine d'années ce genre de processus s'impose, il se peut que les médecins commencent à « oublier » eux-mêmes les conclusions qu'il faudrait tirer en s'appuyant presqu'uniquement sur

les résultats proposés par le programme. A ce moment, nous pourrions dire que ce dernier ne serait plus uniquement un programme d'aide à la décision mais un programme presqu'autonome car le médecin aurait perdu une partie de sa faculté à tirer lui-même le diagnostic final.

Si ces projets d'aide à la décision dans le domaine médical commencent à s'imposer, il faudra alors étudier de manière plus approfondie l'impact que ceux-ci pourraient avoir sur les spécialistes de la médecine et l'influence qu'ils pourraient apporter à leurs verdicts.

Cependant, le cas que nous venons d'évoquer ne devrait pas directement concerner le projet ORTHOGEN dont les limites sont bien fixées avec l'hôpital de Mont-Godinne et qui ne devrait être utilisé uniquement comme un outil de décision, voire un aide-mémoire pour le spécialiste de la médecine.

## 2.2 But du mémoire

Le but de notre mémoire ne couvre pas l'entièreté de ce que nous avons discuté dans la section précédente. Comme nous l'avons déjà signalé, ce travail vient s'intégrer au sein de l'ensemble du processus du projet ORTHOGEN devant amener à créer un outil d'aide à la décision pour les infections orthopédiques. Cependant, nous n'avons pas produit un logiciel directement délivrable tel quel à l'hôpital de Mont-Godinne.

Notre travail consistait en deux objectifs devant aider dans le cadre du projet ORTHOGEN. Ces deux objectifs étaient d'une part de pouvoir manipuler plus facilement la connaissance contenue dans l'ontologie afin de faciliter les dialogues avec les experts médicaux et, d'autre part, de choisir quel moteur d'inférence probabiliste était le plus efficace suivant le contexte d'utilisation de l'outil.

Un premier problème s'était posé à nous lorsque nous avons voulu mettre en relations les deux principales technologies utilisées par le projet, à savoir les ontologies et les moteurs d'inférence probabilistes. Le problème était que les éléments de sortie du premier ne correspondaient pas directement aux éléments d'entrée du second. Pour résoudre ce problème, nous avons tout d'abord décidé de transformer la connaissance provenant de l'ontologie en un ensemble de règles provenant de la logique du premier ordre. En effet, lorsque nous associons à celles-ci un poids, elles peuvent être prises telles quelles comme élément d'entrée pour les moteurs d'inférences probabilistes.

Une fois ce premier choix réalisé, il nous fallait maintenant nous attarder sur le choix de ce fameux moteur d'inférence probabiliste. Pour cela, nous en avons présélectionné trois différents.

Tout d'abord, « Alchemy ». Celui-ci est certainement le plus connu et le plus utilisé. Il est d'ailleurs utilisé dans l'hôpital de Mont-Godinne avec lequel le projet ORTHOGEN travaille. Nous nous devions donc de l'envisager.

Ensuite, « Tuffy ». Ce dernier est également réputé et se vante d'ailleurs d'obtenir de meilleures performances que le premier. Il nous fallait alors vérifier si ces affirmations étaient correctes et surtout dans quels cas elles l'étaient.

Enfin, nous avons également choisi « RockIt ». Il s'agit d'un moteur d'inférence probabiliste qui vient de voir le jour. Il est d'ailleurs toujours en phase de développement. La rédaction de se mémoire se base sur la version 0.3 de celui-ci. Celle-ci ne comporte d'ailleurs pas encore toutes les fonctionnalités car le processus d'apprentissage n'a pas encore été implémenté. Cependant, ayant lu que les résultats retournés par cet outil semblaient très performants, nous nous devions de l'intégrer dans nos choix potentiels.

Après avoir choisi les trois moteurs d'inférence probabiliste que nous allions tester et après avoir défini la procédure qui nous permettrait de faire communiquer l'ontologie à ceux-ci, il nous fallait à présent créer l'outil permettant de réaliser ce travail. Celui-ci devait être composé de plusieurs parties obligatoires.

Tout d'abord, la transformation de la connaissance d'une ontologie en un ensemble de règles provenant de la logique du premier ordre. C'est par ailleurs ces règles que les membres du projet ORTHOGEN devront manipuler afin de tester si la connaissance contenue dans l'ontologie pourrait être améliorée. Notre outil devait alors également proposer un processus permettant à cet utilisateur de facilement modifier ces règles à sa guise afin d'améliorer l'efficacité de ses tests.

En outre, pour faciliter les dialogues avec les médecins concernant la pertinence des règles induites, il nous fallait trouver un processus permettant de simplifier la compréhension de ces règles pour les médecins. En effet, ceux-ci n'étant pas informaticiens, il est fort probable qu'ils ne comprennent pas la logique du premier ordre. Nous avons ainsi décidé d'intégrer, au sein de notre outil, un processus permettant de traduire de manière automatique chaque règle de la logique du premier ordre vers l'anglais. Ainsi, la communication avec les spécialistes médicaux pouvait se faire plus facilement étant donné que le vocabulaire utilisé était commun aux deux parties.

Ensuite, il fallait inclure dans notre programme un moyen de tester l'efficacité en temps et en termes de résultats retournés des trois outils décrits plus haut et d'y produire des statistiques qui permettraient d'améliorer l'analyse de ceux-ci. Ce processus se devait d'être précis étant donné l'importance du choix du moteur d'inférence probabiliste. Comme nous l'avons évoqué précédemment, un des enjeux majeurs de ce projet est la précision des résultats retournés, il nous

fallait alors déterminer quel outil permettait dans quel cas d'obtenir la meilleure précision.

En plus de ces fonctionnalités obligatoires, nous voulions vraiment que l'utilisateur de notre outil ait le plus de facilité possible à effectuer les différentes tâches qui lui incombaient. De ce fait, nous avons le plus souvent possible rajouté des petites fonctions destinées à aider ce dernier. En plus du côté « user-friendly » important dans l'ensemble des outils informatiques existants, la facilité d'emploi de notre programme était primordiale étant donné la difficulté des tâches à réaliser par l'utilisateur de celui-ci et les implications désastreuses d'une erreur de sa part.

En essayant d'être le plus ouvert possible aux désirs de l'utilisateur quant à notre outil, nous avons réussi à nous abstraire d'un contexte précis d'utilisation. Cette abstraction engendre des effets intéressants car elle permet à notre programme de ne pas uniquement posséder une utilité au sein du projet ORTHOGEN mais pourrait également s'étendre à n'importe quel autre domaine de la médecine, voir n'importe quel autre domaine en général. Notre programme permet ainsi de facilement transformer la connaissance contenue dans une ontologie en un ensemble de règles de la logique du premier ordre, d'y effectuer des opérations et ensuite d'obtenir les résultats inférés par ces opérations. L'utilisateur peut utiliser les résultats obtenus comme il le souhaite, mais il peut également utiliser notre fonctionnalité « statistiques » afin de comparer quel moteur d'inférence probabiliste est le plus efficace dans son cas.

Cette abstraction, bien qu'actuellement utilisée dans le domaine des infections orthopédiques, pourrait rendre notre outil plus polyvalent et voir son utilité étendue vers d'autres domaines.

## 2.3 Limite des objectifs

La raison du choix des objectifs liés à ce mémoire s'est faite afin de garder un bon rapport entre le temps passé à développer une fonctionnalité et le gain perçu au niveau, d'une part, de l'efficacité dans la manipulation d'une base de connaissances et, d'autre part, de la précision des statistiques concernant les moteurs d'inférence probabilistes. A cette fin, nous avons parfois été contraints d'abandonner des idées intéressantes mais trop complexes à mettre en place.

Dans son état actuel, notre programme permet à son utilisateur, comme expliqué dans la section 2.2, de réaliser de manière facile et pratique les opérations qu'il souhaite sur l'information contenue dans la base de connaissances d'une ontologie. Il permet également à cet utilisateur d'obtenir des temps d'exécution et une série de résultats retournés par les raisonneurs de la logique de Markov lorsque ceux-ci sont appelés.

Une des premières limites de ce mémoire est le choix de ne pas nous focaliser sur la qualité des résultats obtenus dans notre section statistiques. En effet, il paraît évident que si nous pouvions juger de la qualité d'un résultat, sa pertinence aurait pu alors être déduite plus efficacement. Cependant, pour tester cette qualité, il nous aurait fallu connaître à l'avance quels devraient être les résultats retournés par les moteurs d'inférence probabilistes. Or, cette information n'est généralement pas connue ou alors non dévoilée. Dès lors, nous nous retrouvions avec un grand nombre d'ontologies, mais presqu'aucune d'entre elles ne laissaient figurer quels devaient être les résultats obtenus pour telle ou telle requête. Ainsi, en l'absence de cette information, nous ne pouvions spéculer sur quels étaient les résultats à obtenir à la suite d'une inférence et avons dû nous contenter de nous focaliser sur la partie quantitative de ceux-ci comme il le sera expliqué à la section 3.9.

Une autre limite que nous pourrions attacher à notre mémoire est l'idée d'extraire des données directement des bases de données de l'hôpital de Mont-Godinne. En effet, ce processus aurait été intéressant si l'on souhaitait réaliser des tests plus nombreux. Ainsi, nous obtenions des résultats existants et corrects qui auraient pu être intégrés dans la partie des évidences de notre programme. Cependant, pour réaliser cet objectif, nous avons rencontré plusieurs problèmes majeurs. Tout d'abord, afin d'utiliser des données réelles – même anonymisées – nous avions besoin d'autorisations particulières de la part de l'hôpital. Or, nous n'avons jamais pu obtenir ces dernières. De plus, si nous voulions nous connecter directement à la base de données de cet organisme, il fallait nous interfacer avec cette dernière. Ainsi, de notre côté, il nous aurait fallu créer un module permettant cette tâche. Cette dernière action n'aurait pas été contraignante. Par contre, il aurait également fallu que les responsables informatiques de l'hôpital fassent de même étant donné que nous n'avions pas accès au système informatique de celui-ci. Cette démarche aurait également pu poser des problèmes de temps. Ainsi, pour ces deux raisons, nous avons été obligés d'abandonner cette idée qui aurait pourtant pu nous être très utile pour notre phase de tests.

Enfin, la dernière option que nous avons envisagé d'inclure dans notre programme est la possibilité de se connecter directement à la base de données des moteurs d'inférence probabilistes utilisant cette technologie. Ainsi, dans le cas d'Alchemy, cela n'est pas d'actualité. Cependant, nous aurions pu envisager de reprendre une partie du code des deux autres outils et de nous connecter à leurs bases de données respectives afin d'améliorer les performances au niveau des temps d'exécution. L'idée générale derrière ce procédé aurait été de ne réaliser qu'une seule fois la phase d'instanciation, de retenir son état pour ensuite éviter de la recalculer aux prochaines exécutions en se connectant directement à la base de données et en insérant toutes les clauses instanciées mémorisées. Le problème que nous avons essentiellement rencontré et qui nous a dissuadé de réaliser ce module a été un problème de temps. En effet, cela nous aurait obligé à réaliser de lourdes modifications, en plus de nous obliger à nous plonger en

profondeur dans la compréhension des codes sources de Tuffy et de RockIt. Or, l'intérêt d'améliorer le temps de ces deux outils était assez restreints étant donné qu'il s'agissait plutôt du troisième outil, Alchemy, qui posait des problèmes au niveau de ces temps d'exécution élevés. Nous verrons cela de manière complète dans la section 4.3 qui concerne la phase d'expérimentation.

Comme nous venons de l'expliquer dans cette partie, nous aurions pu intégrer plusieurs autres modules intéressants dans notre programme. Cependant, nous avons volontairement choisi de ne pas les ajouter car la réalisation de ceux-ci comportaient un trop grand nombre de contraintes. Nous nous sommes dès lors cantonnés aux fonctionnalités les plus importantes permettant de répondre aux besoins des objectifs fixés dans la section 2.2.

# Chapitre 3

# Développement

## Sommaire



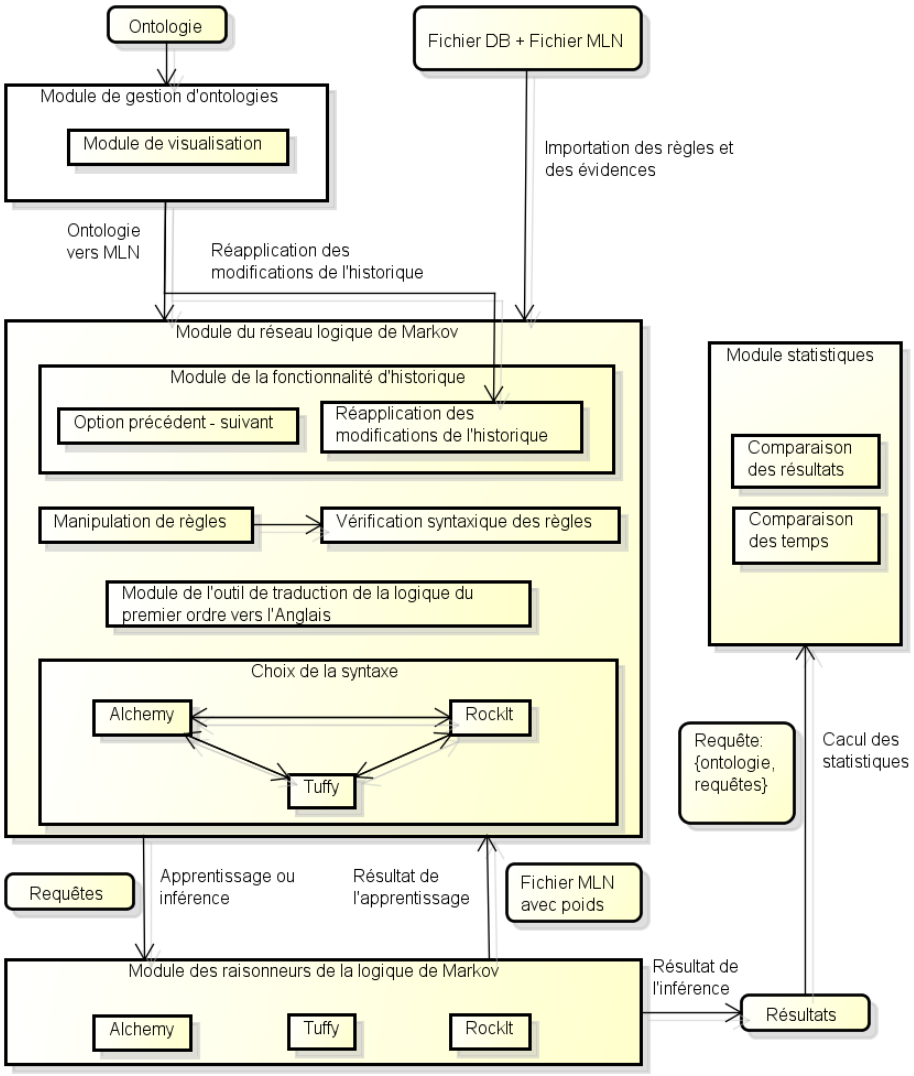
## 3.1 Introduction

Afin de réaliser nos objectifs, nous avons développé un programme comportant environ 25.000 lignes de code. Nous avons volontairement ajouté plusieurs fonctionnalités ne figurant pas directement dans les buts de ce mémoire afin d'améliorer l'utilisabilité et l'expérience de navigation de l'utilisateur.

Initialement, ce programme est destiné aux développeurs du projet ORTHOGEN. Cependant, de par la généralisation des fonctionnalités que nous avons établie, il pourrait être également utilisé dans une multitude d'autres domaines utilisant eux aussi des ontologies et des moteurs d'inférence probabilistes. En outre, l'utilisateur de cet outil peut également utiliser ce dernier comme support afin d'améliorer la communication avec les médecins de l'hôpital de Mont-Godinne ou, le cas échéant, avec une autre personne experte dans le domaine qui est traité par le programme.

L'outil est divisé en plusieurs modules qui sont repris dans le schéma de la figure 3.1.

FIGURE 3.1 – Schéma de l'architecture de notre outil




Dans un premier temps, l'utilisateur doit fournir un moyen d'obtenir un ensemble de règles de la logique du premier ordre sur lequel il va travailler. Pour cette tâche, deux voies sont envisageables.

Premièrement, il peut sélectionner sur son ordinateur une ontologie – dont le format doit nécessairement respecter le standard OWL2 afin d'éviter toute incompatibilité lors de la phase de transformation – et l'ouvrir dans notre l'outil. Cette action entrainera un appel à l'API de Prefuse destinée à offrir une visualisation paramétrable de l'ontologie. Nous reparlerons de cette API dans une section dédiée. Cet outil de visualisation offrira à l'utilisateur une vue plus globale sur la structure de la base de connaissances qui sera ciblée pour la future transformation de l'ontologie. Si cela lui convient, il pourra alors démarrer cette phase de transformation qui aura pour effet de transformer la base de connaissances contenues dans l'ontologie en un ensemble de règles de la logique du premier ordre équivalent. Cet ensemble de règles sera divisé en deux parties distinctes. Tout d'abord, la partie des évidences comportant l'ensemble des évidences contenues dans l'ontologie et enregistrée dans un fichier dont l'extension est .DB. Ensuite, les règles à proprement parler provenant des différentes classes et propriétés contenues dans l'ontologie. Celles-ci comporteront plusieurs sous parties permettant de délimiter la partie des déclarations – dont le seul but est de déclarer les prédicats qui seront utilisés dans les clauses logiques et dans les évidences – des règles logiques. Enfin, tout cet ensemble sera quant à lui enregistré dans un fichier dont l'extension est cette fois-ci .MLN. Le choix de ces deux extensions n'a pas été choisi de manière fortuite. Il s'agit en effet des extensions des deux fichiers d'entrée attendus par les trois raisonneurs de la logique de Markov.

Deuxièmement, dans le cas où l'utilisateur désirerait reprendre un travail déjà commencé plus tôt, il a la possibilité de sélectionner le nom de son fichier pour directement obtenir ces deux ensembles de règles dans l'état où il les avait laissés.

Une fois toutes les règles obtenues dans la logique du premier ordre, l'utilisateur dispose de plusieurs possibilités lui permettant de manipuler celles-ci afin d'obtenir facilement les résultats qu'il souhaite. Il peut effectuer à sa guise des opérations d'ajout, de modification ou de suppression afin d'altérer les futurs résultats calculés par les moteurs d'inférence probabilistes.

Afin de lui éviter toute erreur lors d'exécutions ultérieures, chacun des changements de l'utilisateur est accompagné d'une vérification syntaxique permettant de s'assurer qu'il n'a commis aucune erreur dans sa saisie et que l'opération qu'il vient d'effectuer est bien permise. Ainsi, nous nous assurons à tout moment que les deux fichiers – le fichier .DB et le fichier .MLN – possèdent toujours une syntaxe admise par chacun des moteurs d'inférence probabilistes.

De plus, afin de permettre d'effectuer des tests sur des raisonneurs de la logique de Markov n'acceptant pas la même syntaxe, nous avons ajouté une fonctionnalité permettant le passage d'un de ces outils à un autre en modifiant de manière automatique la syntaxe des règles pour qu'elle soit correctement

comprise par l'outil sélectionné. Cette fonction de notre programme libère l'utilisateur d'une lourde charge de travail qui le forceraient à effectuer un grand nombre de modifications pour tester le même ensemble de règles sur les trois outils différents.

Nous avons ensuite doté notre programme d'une fonctionnalité de traduction automatique. Celle-ci est utile lors de la communication avec les médecins ou avec d'autres experts dans un domaine en particulier. Elle va en l'occurrence transformer les règles écrites dans la logique du premier ordre vers la langue anglaise. Cette action améliore la communication car, dès lors, les discussions ne se font plus sur un langage mathématique qui pourrait ne pas être très compréhensible pour un interlocuteur non formé mais sur le langage naturel compréhensible par tout le monde.

La dernière fonctionnalité de ce module est celle d'historique. Celle-ci est composée en différentes sous parties. Tout d'abord, la fonctionnalité habituelle des options précédent/suivant. Celle-ci comporte toutefois une particularité car elle permet également de réaliser ces opérations après fermeture puis la réouverture de l'outil. Ensuite, la deuxième sous partie de cette fonctionnalité est la capacité à réappliquer automatiquement l'ensemble des changements réalisés sur une certaine version d'une ontologie vers une autre version de cette même ontologie. Ainsi, des utilisateurs gérant des ontologies en constante évolution, comme c'est actuellement le cas pour le projet ORTHOGEN, se verront à nouveau libéré d'une grosse charge de travail car ils ne devront plus réappliquer des opérations qu'ils auraient déjà réalisées antérieurement. Le choix de l'application automatique de ces modifications déjà réalisées se fait dans la phase décrite plus haut qui consiste à transformer l'ontologie en un ensemble de règles. L'utilisateur peut lors de celle-ci choisir, dans le cas d'une ontologie déjà manipulée, s'il souhaite la transformer telle quelle ou alors réappliquer les anciennes opérations réalisées sur celle-ci.

Le module suivant consiste en l'application d'une phase d'apprentissage ou d'inférence par l'un des trois raisonneurs de la logique de Markov sur l'ensemble de règles que l'utilisateur a manipulé. Ce dernier a la possibilité d'effectuer des requêtes sur chacun des moteurs d'inférence probabilistes. Celle-ci comporte le choix de l'outil, la ou les requêtes sur lesquelles porteront l'apprentissage ou l'inférence et, enfin, le choix d'effectuer cette requête pour une phase d'apprentissage ou d'inférence. Lorsque l'outil sélectionné aura terminé son travail, notre programme va, d'une part, enregistrer les résultats obtenus dans un fichier particulier et d'autre part, il va les afficher à l'utilisateur.

Le dernier module de l'outil que nous avons réalisé voit son utilité lorsque plusieurs résultats ont été obtenus par différents moteurs d'inférence probabilistes. En effet, dès lors, l'utilisateur peut vouloir comparer les résultats obtenus, ainsi que les temps ayant été nécessaires pour les obtenir. Nous mettons à sa

disposition deux sous-modules de comparaisons des résultats et des temps obtenus. Pour cela, il peut faire varier plusieurs paramètres – le choix de l'ontologie, la ou les requêtes sélectionnées, ainsi que le choix des résultats obtenus lors de la phase d'apprentissage ou d'inférence – pour obtenir des tableaux détaillés concernant son choix et ainsi lui permettre de tirer les meilleures conclusions quant à l'outil à utiliser dans tel ou tel cas.

## 3.2 Architecture et choix d'implémentation

Pour réaliser notre outil, nous avons opté pour la plateforme Java et plus précisément l'utilisation de l'édition Java SE 1.6. Les raisons de ce choix sont diverses. Tout d'abord, cette plateforme étant très utilisée dans le monde, elle permet une interopérabilité améliorée avec d'éventuels autres logiciels. Ensuite, étant très connue, il existe de nombreux plugins s'intégrant directement et facilement dans des projets écrits en Java. Nous voulions ainsi mettre toutes les chances de notre côté dans l'espoir de récupérer certains modules déjà existants et par conséquent réduire notre charge de travail. Enfin, la dernière raison qui nous a poussés vers ce choix est le fait que deux des trois outils – en l'occurrence, Tuffy et RockIt – soient eux-mêmes implémentés en Java. Réaliser notre projet dans ce langage pourrait ainsi être un plus si nous décidions un jour d'intégrer directement le code de ces deux outils au sein du notre ou encore si nous décidions de nous connecter directement à l'un des outils comme nous l'avons évoqué dans la section 2.3.

Concernant l'architecture de notre projet, nous avons utilisé le design pattern MVC – modèle, vue, contrôleur – afin de séparer notre code en trois couches d'abstraction bien distinctes. Les raisons de ce choix ont fortement été encouragées par la méthodologie de développement itératif qui nous obligeait à développer selon un processus d'évolution continue. En séparant les interfaces du code métier de la sorte, cela nous faisait gagner un temps considérable lors des changements et ajouts qui pouvaient apparaître d'une version à une autre.

En plus de l'utilisation de ce design pattern, nous avons séparé le code de notre projet en différents packages.

Tout d'abord, un package reprenant les interfaces utilisateurs. Celui-ci correspondant à la vue du design pattern MVC décrit plus haut.

Ensuite, le contrôleur se trouve lui aussi dans un package qui lui est propre. Celui-ci contient une seule classe dont les méthodes sont chargées de faire la liaison entre la vue et le modèle.

Enfin, le modèle quant à lui est composé d'un certain nombre de packages classés selon leur rôle dans le programme.

Premièrement, nous avons rassemblé tous les JavaBeans dans un même package. Ces JavaBeans consistent en de simples objets Java respectant la spécification JavaBean d'Oracle [1].

Nous avons ensuite un autre package comportant les classes permettant de transformer la base de connaissances contenue dans une ontologie vers un ensemble de règles de la logique du premier ordre. Cette section est également divisée en un ensemble de sous divisions responsables chacune d'un aspect de cette transformation d'ontologie.

Le package suivant comporte la transformation de règles s'appuyant sur une grammaire écrite en antlr3 [2]. Celui-ci bien que paraissant similaire au package évoqué au paragraphe précédent n'est en réalité pas du tout semblable de par son procédé qui diffère drastiquement. Nous verrons dans les sections qui leur seront dédiées que transformer une ontologie ou une saisie textuelle ne peut se réaliser de la même manière. En outre, la grammaire antlr3 a été choisie pour les mêmes raisons que la plateforme Java : Une grande communauté utilise celle-ci et deux des trois raisonneurs de la logique de Markov que nous utilisons l'utilise également. Ces deux-ci sont à nouveau Tuffy et RockIt.

Nous avons également réalisé une API qui sera décrite plus en détails à la section suivante. Celle-ci comporte également un package qui lui est propre dans lequel se trouve le code de son interface et celui de son implémentation.

Enfin, un dernier package contient l'ensemble des classes utilitaires de notre projet. Parmi celles-ci, nous pouvons retrouver le mécanisme de gestion des fichiers, l'utilisation de loggers ou encore le traitement de properties.

Concernant ces properties, nous pourrions les ajouter à nos choix d'implémentations car celles-ci sont nécessaires afin d'indiquer l'environnement dans lequel va travailler l'utilisateur (Windows, Linux ou MacOs). La spécification de celui-ci est très importante car, dans le cas d'Alchemy, le système d'exploitation joue un rôle sur la façon dont va être réalisée l'exécution de cet outil. En effet, ce moteur d'inférence probabiliste ne fonctionnant pas sous Windows, nous faisons appel à un plugin particulier dans le cas d'une exécution à partir de ce système d'exploitation afin d'émuler une plateforme adéquate. Ces properties sont également requises afin de définir l'emplacement des différents raisonneurs de la logique de Markov sur l'ordinateur de l'utilisateur ; Ceux-ci devant être installés au préalable sur son ordinateur. C'est enfin au sein de ces properties que sera spécifié l'emplacement où l'utilisateur désire enregistrer tous ses fichiers de règles et de résultats. C'est d'ailleurs à partir de cet emplacement que notre programme procédera au chargement des fichiers déjà existants.

1. http ://www.oracle.com/technetwork/Java/Javase/documentation/spec-136004.html
2. http ://www.antlr3.org/

Enfin, notre dernier package contient l'ensemble du code source du plugin « Prefuse »[3] que nous avons utilisé dans le cadre d'un outil de visualisation d'ontologies. Ce plugin justifie à nouveau le choix de la plateforme Java car il est lui-même implémenté dans ce langage, ce qui nous a fortement facilité son intégration dans notre programme.

Nous pouvons terminer cette sous-section en disant que nos choix d'implémentation reposent sur un ensemble de technologies répandues, ce qui nous permet, d'une part, de compter sur une communauté étendue en cas de problème et, d'autre part, d'améliorer notre intégration avec d'autres programmes. Nous avons également opté pour une architecture facilitant le développement itératif, évolutif et la maintenance de notre outil car c'est cette méthodologie que nous avons suivie tout au long du déroulement de notre projet.

## 3.3 Application programming interface

Afin d'augmenter l'interopérabilité de notre outil avec d'autres logiciels, nous avons créé une API (Application Programming Interface) proposant les principales fonctionnalités que nous vous décrivons dans ce mémoire.

Cette API a pour objectif d'améliorer la réutilisabilité de nos modules. Ainsi, si nous décidions de migrer vers une plateforme web, nous ne devrions pas tout implémenter à nouveau mais il nous suffirait de nous interfacer avec notre programme pour obtenir ses fonctions. De même, si un autre outil désire obtenir certaines de nos fonctions, il pourrait les utiliser facilement et cela sans même connaître la structure de notre projet.

Nous n'avons bien sûr pas repris dans l'API une liste exhaustive de nos fonctionnalités, mais nous nous sommes cantonnés aux plus importantes et aux plus pertinentes.

A la figure 3.2, nous vous offrons une vue de notre interface et des méthodes qui la composent. Plus bas, nous vous décrirons chacune de celles-ci en détails.

Tout d'abord, nous avons mis à disposition une méthode permettant d'effectuer un apprentissage sur un fichier avec un certain outil. L'exécution engendre une modification des fichiers spécifiés comme lors d'un apprentissage normal de notre outil. Cette méthode retourne un booléen indiquant si l'opération s'est bien effectuée ou si elle a rencontré un problème. En outre, nous devons spécifier en paramètres l'outil, le nom du fichier impliqué qui sera situé à l'emplacement défini par les properties comme expliqué dans la section précédente, ainsi que différentes options possibles dans le cas d'un apprentissage.

3. http ://prefuse.org

FIGURE 3.2 – Vue de l'API

```java
package fundp.info.ac.be.API;

import java.util.ArrayList;

public interface OperationsInterface {

    public boolean runLearning(int tool, String fileName, boolean discriminativeLearning,
                               ArrayList<String> listNonEvidences, boolean windowsOS, String toolLocation);


    public boolean runInfer(int tool, String fileName, ArrayList<String> listQueries, boolean windowsOS, String toolLocation);


    public boolean parseOntology(String fileName, String ontologyPath, String toolLocation, int tool);


    public void mergeFiles(String sourceFileName, String addedFileName, String filesLocation);


    public boolean parseAndLearn(String fileName, String toolLocation, String ontologyPath, int tool,
                                 boolean discriminativeLearning, ArrayList<String> listNonEvidences, boolean windowsOS);

}
```

Deuxièmement, nous proposons une méthode similaire permettant, cette fois, d'effectuer une inférence sur un fichier avec un outil choisi. Le processus est identique et se charge de modifier les fichiers impliqués et de retourner une valeur booléenne de vérification du bon déroulement de l'opération. Les paramètres comportent également l'outil, le nom du fichier et des options similaires à la première méthode décrite mais, cette fois, dans le cadre de l'inférence.

La troisième méthode a pour but de créer une paire de fichiers comportant les extensions .DB et .MLN à partir d'une ontologie dont le chemin est spécifié en paramètre. Il suffit de fournir ce dernier plus l'une ou l'autre information comme le choix du nom du fichier ou encore la syntaxe de l'outil dans laquelle seront écrits les fichiers de destinations et le programme exécutera le processus de transformation d'ontologies décrit à la section 3.4. Enfin, une valeur booléenne de confirmation est également retournée à la fin du déroulement de la fonction.

La quatrième méthode que nous proposons permet de fusionner des fichiers de règles et d'évidences comme il est notamment le cas pour la fonctionnalité qui concerne la réapplication des modifications de l'historique. Celle-ci a besoin de deux paramètres. D'une part, un fichier source contenant les règles et évidences. D'autre part, un fichier comportant des modifications qu'il faudra répercuter sur le fichier source. Cette méthode permet d'ajouter ou de modifier des règles mais également de modifier les poids selon la comparaison entre les deux fichiers. Lorsque l'exécution de cette méthode est terminée, seul le fichier source se voit modifié et aucune valeur n'est retournée à la fin de l'exécution.

Enfin, la dernière méthode ne propose pas de fonctionnalités exclusivement nouvelles. Il s'agit en réalité de la combinaison de la transformation d'une ontologie en règles de la logique du premier ordre, suivie d'un apprentissage. La raison pour laquelle nous avons choisi de réaliser cette méthode est simplement pour fournir une facilité à l'utilisateur. En effet, après discussion avec nos différents responsables, il en est ressorti que ce scénario de test – transformation

de l'ontologie puis apprentissage sur cette même ontologie – était assez fréquent et qu'il fallait alors y apporter quelques facilités d'utilisation.

Nous venons ici de vous présenter notre API, ses fonctionnalités et la raison de son existence. D'autres méthodes auraient pu et pourraient encore voir le jour. Cependant, à l'heure actuelle, nous nous sommes concentrés sur ce qui nous semblait être le principal et l'essentiel des possibilités qu'offre notre outil.

## 3.4 Analyse syntaxique des ontologies

Une des fonctionnalités principales de notre outil est sa capacité à transformer une ontologie – respectant le format OWL2 – en un ensemble de règles et d'évidences écrites dans la logique du premier ordre. Cet ensemble de règles provient de la base de connaissances contenue dans l'ontologie. Les classes et propriétés de celle-ci engendrent des clauses logiques, tandis que les individus qui peuplent cette base de connaissances se transforment en évidences. Outre ces deux aspects, l'ensemble des règles contient également une partie composée de déclarations qui reprend tous les prédicats définis dans les deux parties précédentes.

Afin d'obtenir la transformation de l'ontologie qu'il souhaite, l'utilisateur doit effectuer une série de tâches assez simples.

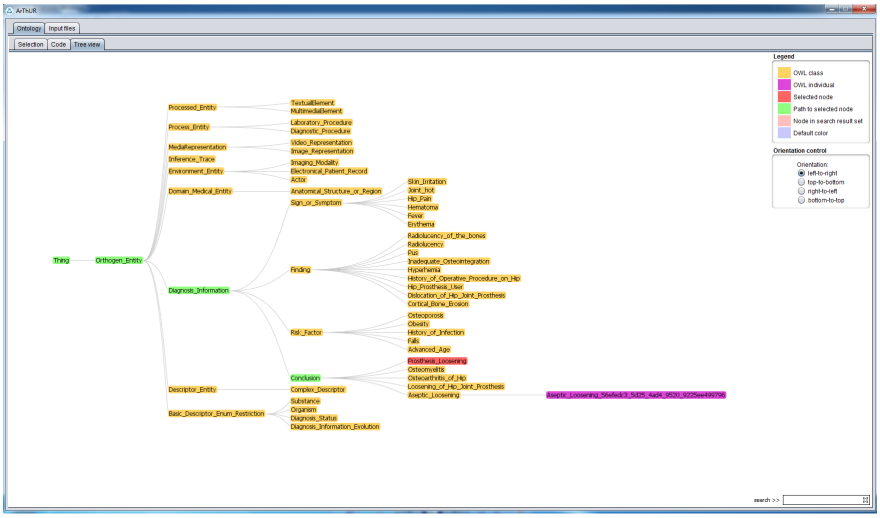
Premièrement, il lui faut sélectionner sur son ordinateur l'ontologie sur laquelle il désire travailler. Pour cela, il lui suffit de cliquer sur le bouton prévu à cet effet et de sélectionner l'emplacement de son ontologie. Une fois celle-ci sélectionné, il lui faut alors l'ouvrir. Cette opération engendre, d'une part, le chargement du code source dans le cas où l'utilisateur voudrait directement se plonger dans le code source de l'ontologie, notamment lorsqu'un problème est constaté. D'autre part, l'utilisateur obtient également un outil de visualisation que nous décrirons plus loin.

Lorsque l'ontologie a été chargée, l'utilisateur peut maintenant nommer l'ensemble de fichiers qui résultera de la transformation. Il peut également choisir sur quelle partie de l'ontologie il travaillera. S'il ne désire pas travailler sur l'ensemble de l'ontologie, il lui suffit de cocher l'option correspondante. Une nouvelle fenêtre apparaît alors lui offrant la possibilité de choisir la ou les racines qu'il souhaite inclure à sa transformation. Une fois cette sélection effectuée, une dernière fenêtre lui offrira la possibilité d'inclure ces racines – de ne prendre que les racines sélectionnées et leurs fils – ou d'exclure celles-ci – de ne choisir que ses pères. Après ce processus, les évidences et règles, ainsi que leurs fichiers correspondant seront créés et affichés sur l'écran de l'utilisateur.

L'ensemble du processus décrit ci-dessus est repris dans les figures A.2 à A.5 situées en annexe.

Afin de permettre à l'utilisateur de choisir de travailler sur une sous partie de l'ontologie sélectionnée – s'il souhaite, par exemple, comparer l'évolution du temps au fur et à mesure de l'apprentissage de règles provenant d'une ontologie plus conséquente comme nous le ferons dans la section expérimentation – nous avons doté notre outil d'un moteur de visualisation permettant de naviguer parmi les différentes branches de cette ontologie. Une représentation de cet outil de visualisation se trouve à la figure 3.3

FIGURE 3.3 – Moteur de visualisation d'ontologies




Cet outil de visualisation possède un code couleur permettant de différencier les classes des évidences et de repérer les sous-classes et super-classes de la sélection actuelle de l'utilisateur. L'outil de visualisation affiche également les évidences en mauve afin que l'utilisateur puisse les identifier facilement. Il comprend également un moteur de recherche permettant de retrouver plus facilement une classe ou évidence précise au sein de l'arborescence.

Pour réaliser cette tâche, nous nous sommes appuyés sur une API proposée par l'organisme Prefuse à laquelle nous avons apporté certaines modifications afin de permettre son intégration au sein du processus de notre outil. Cependant, étant donné que ce plugin était développé tout comme notre projet en Java, nous n'avons pas rencontré de problème portant à conséquence lors de l'intégration.

Ainsi, lorsque l'utilisateur a choisi une ontologie sur son ordinateur, il obtient une visualisation de celle-ci accompagnée du code source de l'ontologie. Il dispose par conséquent d'un grand nombre d'outils afin de l'aider à choisir judicieusement la façon dont il utilisera cette dernière.

Une fois que l'utilisateur a réalisé tous ses choix et qu'il souhaite transformer l'ontologie en règles de la logique du premier ordre, notre outil va entreprendre un processus qui transformera le code écrit en OWL2 en cet ensemble de règles.

Pour réaliser ces transformations, nous nous sommes appuyés sur l'API org.semanticweb.owl [4]. Celle-ci offre l'avantage de contenir une large gamme de méthodes facilitant la manipulation des ontologies à partir du langage Java. Ces facilités vont de l'ouverture de l'ontologie, à la récupération de ses classes et propriétés en passant par des fonctions de transformations du langage OWL2 vers la logique du premier ordre. Chaque axiome a ainsi pu être transformé facilement en suivant les règles de transformations de la W3C reprises dans les figures 3.4 à 3.6 [5].

Lorsque nous avons obtenu l'ensemble des règles qui composaient l'ontologie, nous leur avons associé un poids initial de 1.0. Nous avons choisi cette convention par défaut car les outils que nous testerons par la suite nécessitent pour la plupart que les règles possèdent un poids initial et cela même avant de commencer la phase d'apprentissage. Le choix du poids de 1.0 restait relativement neutre quant au calcul des probabilités, il s'agit du poids par défaut. Cela ne biaise donc pas les tests initiaux.

Une fois toutes ces étapes réalisées, nous avons exécuté notre script permettant de transformer les règles de la logique du premier ordre selon la syntaxe de l'outil sélectionné. Cette dernière étape apparemment redondante est essentielle afin de permettre à notre ensemble de règles de respecter la syntaxe de l'outil choisi tel qu'on peut le voir dans la figure A.3. Chaque outil comportant une syntaxe qui lui est propre et qui n'est pas toujours compatible avec celle des autres, il fallait s'assurer que celle-ci soit bien respectée.

Cette dernière étape réalisée, nous pouvions afficher toutes ces règles dans notre outil afin de permettre à l'utilisateur d'effectuer les opérations qu'il souhaitait concernant ces dernières. Les prochaines sections vont notamment vous expliquer comment l'utilisateur peut alors gérer et utiliser les règles qu'il vient d'obtenir.

## 3.5 Changement de la grammaire en fonction du moteur d'inférence probabiliste

Une des fonctionnalités les plus puissantes de notre programme est le passage d'un outil à un autre de manière automatique en un clic.

4. http ://semanticweb.org/wiki/OWL_API
5. http ://www.w3.org/TR/owl2-syntax/

FIGURE 3.4 – Transformations de OWL2 vers la logique du premier ordre

## OWL2 interpretation as FOL

| Class Expressions | FOL | Example | Translation |
|---|---|---|---|
| **Propositional Connectives and Nominals** | | | |
| IntersectionOf($CE_1$,...,$CE_n$) | $\forall x : CE_1(x) \wedge ... \wedge CE_n(x)$ | IntersectionOf(A, IntersectionOf(B,C)) | $\forall x : A(x) \wedge (B(x) \wedge C(x))$ |
| UnionOf($CE_1$,...,$CE_n$) | $\forall x : CE_1(x) \vee ... \vee CE_n(x)$ | UnionOf(A, IntersectionOf(B,C)) | $\forall x : A(x) \vee (B(x) \wedge C(x))$ |
| ComplementOf($CE_1$) | $\forall x : \neg CE_1(x)$ | ComplementOf(A) | $\forall x : \neg A(x)$ |
| OneOf($a_1$,...,$a_n$) | $a_1 \vee ... \vee a_n$ | OneOf(a,b,c) | $a \vee b \vee c$ |
| **Object Property Restrictions** | | | |
| SomeValuesFrom(OPE,CE) | $\forall x, \exists y : OPE(x,y) \wedge CE(y)$ | SomeValuesFrom(P,ComplementOf(A)) | $\forall x, \exists y : P(x,y) \wedge \neg A(y)$ |
| AllValuesFrom(OPE,CE) | $\forall x, y : OPE(x,y) \Rightarrow CE(y)$ | AllValuesFrom(P,UnionOf(A,B)) | $\forall x, y : P(x,y) \Rightarrow A(y) \vee B(y)$ |
| HasValue(OPE,a) | $\forall x : OPE(x,a)$ | HasValue(P,a) | $\forall x : P(x,a)$ |
| ExistsSelf(OPE) | $\forall x : OPE(x,x)$ | ExistsSelf(P) | $\forall x : P(x,x)$ |
| **Object Property Cardinality Restrictions** | | | |
| MinCardinality(n,OPE,CE) | $\exists y_1, ..., y_n : \Lambda_{1 \le i \le n}(OPE(x,y_i) \wedge CE(y_i)) \wedge \Lambda_{1 \le i < n, i < j \le n} y_i \neq y_j$ | MinCardinality(2,P,A) | $\exists y_1, y_2 : (P(x,y_1) \wedge A(y_1)) \wedge (P(x,y_2) \wedge A(y_2)) \wedge y_1 \neq y_2$ |
| MaxCardinality(n,OPE,CE) | $\forall y_1, ..., y_{n-1} : \bigwedge_{1 \le i \le n-1} (OPE(x,y_i) \wedge CE(y_i)) \Rightarrow \bigvee_{1 \le i < n-1, i < j < n} y_i = y_j$ | MaxCardinality(3,P,A) | $\exists y_1, y_2 : (P(x,y_1) \wedge A(y_1)) \wedge (P(x,y_2) \wedge A(y_2)) \Rightarrow y_1 = y_2$ |

FIGURE 3.5 – Transformations de OWL2 vers la logique du premier ordre (2)

| Axioms | FOL | Example | Translation |
|---|---|---|---|
| **Class Expression Axioms** | | | |
| SubClassOf($CE_1$, $CE_2$) | $\forall x : CE_1(x) \Rightarrow CE_2(x)$ | SubClassOf(A, IntersectionOf(B,C)) | $\forall x : A(x) \Rightarrow (B(x) \wedge C(x))$ |
| EquivalentClasses($CE_1$,...,$CE_n$) | $_{1 \le i \le n-1}\forall x : CE_i(x) \Leftrightarrow CE_{i+1}(x)$ | EquivalentClasses(A, UnionOf(B,C),D) | $\forall x : A(x) \Leftrightarrow B(x) \vee C(x)$<br>$\forall x : B(x) \vee C(x) \Leftrightarrow D(x)$ |
| DisjointClasses($CE_1$,...,$CE_n$) | $_{1 \le i < n, i < j \le n}\forall x : CE_i(x) \Rightarrow \neg CE_j(x)$ | DisjointClasses(A, UnionOf(B,C),D) | $\forall x : A(x) \Rightarrow \neg(B(x) \vee C(x))$<br>$\forall x : A(x) \Rightarrow \neg D(x)$<br>$\forall x : B(x) \vee C(x) \Rightarrow \neg D(x)$ |
| **Object Property Axioms** | | | |
| SubPropertyOf($OPE_1$,$OPE_2$) | $\forall x, y : OPE_1(x,y) \Rightarrow OPE_2(x,y)$ | SubPropertyOf(P,Q) | $\forall x, y : P(x,y) \Rightarrow Q(x,y)$ |
| EquivalentProperties($OPE_1$,...,$OPE_n$) | $_{1 \le i \le n-1}\forall x, y : OPE_i(x,y) \Leftrightarrow OPE_{i+1}(x,y)$ | EquivalentProperties(P,Q,R) | $\forall x, y : P(x,y) \Leftrightarrow Q(x,y)$<br>$\forall x, y : Q(x,y) \Leftrightarrow R(x,y)$ |
| DisjointProperties ($OPE_1$,...,$OPE_n$) | $_{1 \le i < n, i < j \le n}\forall x : OPE_i(x,y) \Rightarrow \neg OPE_j(x,y)$ | DisjointProperties(P,Q,R) | $\forall x, y : P(x,y) \Rightarrow \neg Q(x,y)$<br>$\forall x, y : P(x,y) \Rightarrow \neg R(x,y)$<br>$\forall x, y : Q(x,y) \Rightarrow \neg R(x,y)$ |
| ObjectPropertyDomain(OPE,CE) | $\forall x, y : OPE(x,y) \Rightarrow CE(x)$ | ObjectPropertyDomain(P,ComplementOf(A)) | $\forall x, y : P(x,y) \Rightarrow \neg A(x)$ |
| ObjectPropertyRange(OPE,CE) | $\forall x, y : OPE(x,y) \Rightarrow CE(y)$ | ObjectPropertyRange(P,UnionOf(A,B)) | $\forall x, y : P(x,y) \Rightarrow A(y) \vee B(y)$ |
| InverseProperties($OPE_1$,$OPE_2$) | $\forall x, y : OPE_1(x,y) \Rightarrow OPE_2(y,x)$ | InverseProperties(P,Q) | $\forall x, y : P(x,y) \Rightarrow Q(y,x)$ |
| FunctionalProperty(OPE) | $\forall x, y, z : OPE(x,y) \wedge OPE(x,z) \Rightarrow y = z$ | FunctionalProperty(P) | $\forall x, y, z : P(x,y) \wedge P(x,z) \Rightarrow y = z$ |
| InverseFunctionalProperty(OPE) | $\forall x, y, z : OPE(y,x) \wedge OPE(z,x) \Rightarrow y = z$ | InverseFunctionalProperty(P) | $\forall x, y, z : P(y,x) \wedge P(z,x) \Rightarrow y = z$ |
| ReflexiveProperty(OPE) | $\forall x, y : OPE(x,y) \Rightarrow OPE(x,x)$ | ReflexiveProperty (P) | $\forall x, y : P(x,y) \Rightarrow P(x,x)$ |

FIGURE 3.6 – Transformations de OWL2 vers la logique du premier ordre (3)

| | | | |
|---|---|---|---|
| IrreflexiveProperty(OPE) | $\forall x, y : OPE(x, y) \Rightarrow \neg OPE(x, x)$ | IrreflexiveProperty (P) | $\forall x, y : P(x, y) \Rightarrow \neg P(x, x)$ |
| SymmetricProperty(OPE) | $\forall x, y : OPE(x, y) \Leftrightarrow OPE(y, x)$ | SymmetricProperty (P) | $\forall x, y : P(x, y) \Leftrightarrow P(y, x)$ |
| AsymmetricProperty(OPE) | $\forall x, y : OPE(x, y) \Leftrightarrow \neg OPE(y, x)$ | AsymmetricProperty (P) | $\forall x, y : P(x, y) \Leftrightarrow \neg P(y, x)$ |
| TransitiveProperty(OPE) | $\forall x, y, z : OPE(x, y) \wedge OPE(y, z) \Rightarrow OPE(x, z)$ | TransitiveProperty (P) | $\forall x, y, z : P(x, y) \wedge P(y, z) \Rightarrow P(x, z)$ |
| **Assertions** | | | |
| ClassAssertion(CE,a) | $CE(a)$ | ClassAssertion(IntersectionOf(A,B),a) | $A(a) \wedge B(a)$ |
| PropertyAssertion(OPE,a,b) | $OPE(a, b)$ | PropertyAssertion(P,a,b) | $P(a, b)$ |
| NegativePropertyAssertion(OPE,a,b) | $\neg OPE(a, b)$ | PropertyAssertion(P,a,b) | $\neg P(a, b)$ |
| SameIndividual($a_1$,..., $a_n$) | - | - | - |

| Symbol | Meaning |
|---|---|
| **CE** | Class Expression |
| **a,b,c,...** | Individual |
| **OPE** | Object Property Expression |
| **n** | Non Negative Integer |

Un des majeurs problèmes que nous avons rencontrés lors de notre développement a été la différence entre les syntaxes acceptées par les différents outils que nous devions manipuler pour obtenir nos résultats. En effet, la moindre erreur empêchait tout simplement l'exécution du processus de calcul et par conséquent l'obtention des résultats. L'un des buts de notre mémoire étant la facilité d'utilisation de notre outil, nous ne voulions pas contraindre l'utilisateur à ne travailler qu'avec un seul raisonneur de la logique de Markov choisi préalablement pour chaque transformation d'ontologie. Cela l'aurait contraint à devoir répéter trois fois le même processus pour tester la différence entre l'exécution des trois outils. Cette démarche redondante est bien sûr à éviter si l'on veut que notre programme puisse être utilisé dans les meilleures conditions.

C'est pour cela que nous avons entrepris d'introduire des changements inter outils sans que cela ne nécessite d'effort particulier de la part de l'utilisateur.

La mise en place – qui n'a pas été aisée – a pu se faire grâce à un arbre syntaxique réalisé en Java accompagné par une grammaire écrite dans la technologie antlr3. Comme nous l'avons déjà évoqué, le choix de ce framework peut se justifier par le fait que deux des trois outils utilisent également antlr3. Dans une démarche d'évolution, nous pensons ainsi que le choix de cette technologie est le plus intéressant.

Le principe de cette grammaire est simple. Nous y décrivons l'ensemble des éléments qui composent les règles et nous les combinons ensemble de manière récursive. Cette description va des prédicats, aux disjonctions et autres opérateurs, en passant par les poids pour arriver à l'élément racine comportant l'entièreté des règles.

Nous veillons également à ce que les règles de cette grammaire se veuillent le plus générales possibles. Ainsi, un bon exemple serait les quantificateurs existentielles qui pourraient s'écrire de 5 façons différentes prenant en compte les

différentes orthographes possibles du mot ou encore la différence entre la syntaxe proposée par RockIt et celle proposée par les deux autres outils.

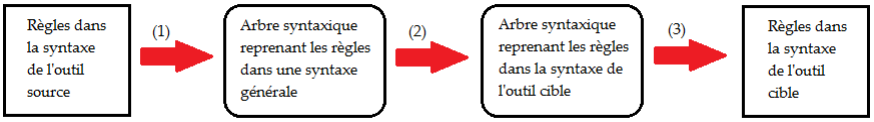
Hormis la composition de la grammaire, nous ajoutons pour chaque règle de celle-ci du code applicatif Java destiné à construire un arbre syntaxique. Celui-ci sera, dans le cas où la grammaire accepte l'entrée, parcouru à l'aide d'un algorithme récursif afin d'effectuer les différentes transformations nécessaires pour passer d'un moteur d'inférence probabiliste à un autre.

Lors du parcours de cet arbre, une fois que ce dernier aura été construit grâce au framework antlr3, une série de paramètres seront à prendre en compte. Tout d'abord, il est important de savoir quel sera le raisonneur cible. Le raisonneur source n'a quant à lui aucune importance étant donné que les règles de notre programme vont transiter dans la grammaire et que cette action est réalisée de la même manière quel que soit leur origine.

L'outil cible a cependant bien sûr de l'importance afin de savoir quelle sera la nouvelle syntaxe à respecter après que les transformations aient été effectuées. Une fois celui-ci déterminé, l'algorithme dédié va être effectué. Il sera chargé de transformer récursivement chaque élément qui compose les règles de la même manière que la grammaire antlr3 le réalisait pour créer l'arbre initial.

Une fois cette dernière étape réalisée, il ne restera plus qu'à afficher les changements à l'utilisateur qui n'aura eu d'autre travail que de cliquer sur le nouvel outil souhaité. Un récapitulatif du processus réalisé lors d'un changement de grammaire est repris à la figure 3.7.

FIGURE 3.7 – Processus du changement d'outil




La puissance de ce processus est telle que nous l'utilisons également afin de vérifier si l'ajout et la modification de règles n'engendrent pas de problèmes. La création de l'arbre syntaxique est entièrement similaire. Seule la partie du programme Java utilise un algorithme différent. Celui-ci est chargé d'une part de vérifier si la vérification syntaxique du compilateur n'a pas amené à une erreur et, d'autre part, si l'ajout et la modification de règles n'implique pas un non respect des conventions syntaxiques d'écriture des règles. Nous reparlerons de ce problème dans la section suivante.

La traduction des règles utilise également ce processus. En effet, la tokenisation de celles-ci permet une gestion beaucoup plus simple par le programme Java qui en est responsable. Nous obtenons directement la structure composée des différents opérateurs de la logique du premier ordre à laquelle nous ajoutons les prédicats associés à ces opérateurs. Une fois cette structure obtenue, il nous est beaucoup plus simple de traduire les règles en les décomposant récursivement. Nous expliciterons ce processus plus en détails dans la section 3.7. La décomposition de ces règles grâce à l'arbre syntaxique permet ainsi de pouvoir plus facilement gérer la manière dont celles-ci seront traduites.

Enfin, ce processus se retrouve également à la fin du script permettant le passage d'une ontologie en règles de la logique du premier ordre. Ceci nous permet de nous abstraire de toute syntaxe lorsque nous transformons cette ontologie étant donné, qu'au final, nous modifions de toute façon le résultat obtenu suivant la syntaxe du raisonneur sélectionné. Cela nous permet également d'utiliser de manière plus efficace le plugin dédié aux ontologies expliqué dans la partie précédente car nous ne devons pas produire de syntaxe de manière conditionnelle – suivant l'outil sélectionné –mais plutôt, nous produisons le résultat selon une syntaxe générale qui sera dans tous les cas retransformée en fin de processus.

Nous venons de voir que la grammaire antlr3 et les différents processus écrits en Java dans le code applicatif offrent une puissance de calcul importante à plusieurs de nos fonctionnalités. L'élaboration de celle-ci a donc été un énorme plus pour le côté « user-friendly » de notre application car l'utilisateur pouvait alors s'abstraire de l'utilisation d'une syntaxe en particulier lors de son développement.

## 3.6 Manipulation de règles et d'évidences

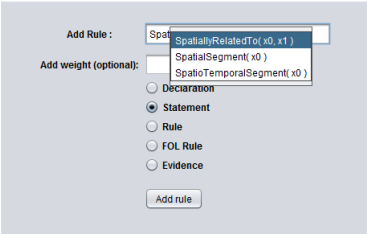
Une des fonctionnalités les plus intéressantes de notre outil est sa puissance à manipuler différentes règles obtenues à partir d'une ontologie. En effet, il ne se contente pas uniquement de transformer cette dernière en un ensemble de règles provenant de la logique du premier ordre, il permet également une gestion complète de celles-ci.

Ainsi, l'utilisateur a la possibilité de modifier cet ensemble à sa guise afin d'altérer les différents scénarios de tests.

Tout d'abord, notre outil offre les fonctionnalités d'ajout, de modification et de suppression de règles. Celles-ci, bien que triviale, comportent déjà un certain nombre de contraintes. Lorsque l'on ajoute ou modifie, le programme doit bien sûr veiller à ce que l'opération de l'utilisateur n'entraine pas un état incohérent de son ensemble de règles. Ainsi, il faut tout d'abord veiller à ce que les prédicats contenus dans la règle ou dans l'évidence que l'on ajoute soient bien présents dans la liste des déclarations.

Afin de permettre plus facilement l'ajout de règles valides, nous avons introduit un mécanisme d'auto-complétion. Un exemple de cette fonctionnalité est repris à la figure 3.8. Le principe de celle-ci est de proposer la liste des prédicats existants commençant par la saisie de l'utilisateur. En plus d'éviter les erreurs de nommage des prédicats, notre programme va afficher le nombre de paramètres que ceux-ci contiennent. Ainsi, grâce à cette facilité, l'utilisateur ne pourrait plus se retrouver dans un scénario où une erreur survient car le prédicat ajouté n'existe pas avec le nombre de paramètres saisis.

FIGURE 3.8 – Mécanisme d'auto-complétion




L'auto-complétion a également un effet pratique pour l'utilisateur au niveau de la recherche. En effet, si celui-ci se trouve dans une situation pour laquelle un grand nombre de déclarations existent, il pourrait ne plus savoir le nom exact du prédicat qu'il cherche à insérer, surtout si ces noms sont particulièrement longs comme c'est parfois le cas dans les grosses ontologies. Ainsi, s'il est repris dans la liste que lui propose notre programme, il pourra facilement le retrouver et cela lui évitera d'insérer des erreurs syntaxiques.

En outre, il est nécessaire d'entreprendre des vérifications en ce qui concerne la syntaxe des règles à ajouter. En effet, les différents outils possédant chacun une syntaxe qui leur est propre, nous devions à tout prix prendre en considération un certain nombre d'aspects :

Premièrement, nous devions penser à respecter la syntaxe imposée par chaque outil. Un bon exemple serait la façon dont est représentée une clause existentielle. Alors qu'Alchemy et Tuffy permettent l'écriture « Exist », RockIt utilise une notation qui lui est entièrement propre composée d'une première partie, placée en début de règle, indiquant les variables ciblées par le quantificateur existentiel et d'une seconde, en fin de règle, indiquant le nombre maximum d'occurences pouvant être présentes. Il n'y a donc pas de notation universelle qui pourrait être comprise par les trois raisonneurs. Outre la syntaxe des clauses, la richesse des opérations possibles s'avère également être un point crucial. RockIt n'accepte, par exemple, pas les implications qui doivent alors être transformées sous forme de disjonctions.

Après nous être préoccupé de la syntaxe, nous devions alors vérifier la pertinence de la règle introduite. Un bon exemple pour illustrer ce principe serait de ne pas permettre l'ajout d'un prédicat si celui-ci n'existe pas avec le bon nombre de paramètres dans les déclarations. Ainsi, « $Eleve(Martin, Mathieu)$ » ne devrait pas être permis si la déclaration de ce prédicat est « $Eleve(Individual)$ ». L'auto-complétion que nous avons décrite plus haut empêche d'introduire ces erreurs, mais elle n'est en réalité qu'une aide pour l'utilisateur qui n'est pas obligé de l'utiliser. Il nous a ainsi fallu penser à d'autres processus pour vérifier la pertinence des règles introduites.

Nous devions ensuite vérifier si la section choisie était bien en adéquation avec le résultat inséré. Ainsi, « $Eleve(Martin) \Rightarrow Etudie(Martin)$ » ne pourrait pas être inséré dans la partie « Déclaration » de notre outil. Cela serait bien sûr incohérent, car il s'agit ici d'une règle et non de la déclaration d'un prédicat.

Nous devions ensuite prendre en compte des spécificités plus techniques qui touchaient aux évidences. Les paramètres qui la composent doivent obligatoirement commencer par une majuscule ou être entourés de guillemets sans quoi certains outils pourraient retourner des erreurs syntaxiques.

Enfin, les poids étaient également importants à gérer. Nous avons empêché les omissions en imposant un poids par défaut ayant comme valeur 1.0 comme nous en avons discuté dans la section 3.4. Cependant, dans certains cas – l'ajout d'une règle comportant une clause existentielle avec l'outil RockIt – le poids ne pouvait pas être déclaré car RockIt permet d'utiliser uniquement le quantificateur existentiel avec un poids infini. Nous avons donc été obligé d'intégrer un script permettant de traiter ce cas particulier.

Pour faciliter l'ensemble de cette gestion, nous avons intégré un module de transformation permettant à l'utilisateur d'insérer une « pseudo règle » qui n'est pas obligée de respecter la syntaxe exacte du langage. Ce degré d'imprécision est admis car notre compilateur va, derrière les coulisses, transformer la saisie de l'utilisateur en une entrée compréhensible par n'importe lequel des trois moteurs d'inférence probabilistes.

La maitrise que l'utilisateur a sur son ensemble de règles ne s'arrête pas là. En effet, grâce à un système d'historique – qui sera explicité dans la section 3.8 – l'utilisateur peut réaliser des scénarios tels que :

- Obtenir des règles à partir d'une ontologie
- Effectuer des modifications au sein de ces règles suivant la syntaxe compatible avec l'outil qu'il a sélectionné
- Réaliser un apprentissage avec l'un des raisonneurs de la logique de Markov
- Obtenir les résultats grâce à une phase d'inférence réalisée avec ce même raisonneur

- Effectuer un retour en arrière afin de revenir juste avant l'étape d'apprentissage
- Changer d'outil
- Répéter les phases d'apprentissage et d'inférence pour ce nouvel outil

De tels scénarios permettent de comparer les résultats de deux outils sans être perturbé par les différents ajouts ou suppressions induits par le premier apprentissage et, surtout, sans obliger l'utilisateur, lorsqu'il change d'outil, de devoir réécrire toutes les étapes réalisées entre la transformation de l'ontologie en règles de la logique du premier ordre et la première phase d'apprentissage.

Une autre partie importante dans la gestion des règles est la suppression de ces dernières. Cette partie bien que paraissant banale engendre toutefois un assez grand nombre de contraintes. Ainsi, lorsque l'on supprime une déclaration, il est essentiel de supprimer les règles et évidences utilisant cette dernière sans quoi les outils retourneraient des erreurs pour cause de prédicat non déclaré. Cependant, il se pourrait que l'utilisateur ne soit pas conscient qu'il reste encore des règles utilisant ce prédicat. Nous avons alors décidé de le notifier le cas échéant. Ainsi, s'il s'avérait que cette suppression soit une erreur, il peut choisir de l'annuler et le programme lui indiquerait en couleur quelles lignes sont concernées par la déclaration qu'il souhaitait supprimer. Les figure A.6 à A.8 reprennent une illustration de ce méchanisme.

Les fonctionnalités que nous venons de présenter dans cette partie devraient ensemble permettre à un utilisateur de réaliser rapidement et facilement les modifications qu'il souhaite afin de pouvoir tester de manière simple et efficace les différents scénarios qui engendreront des résultats.

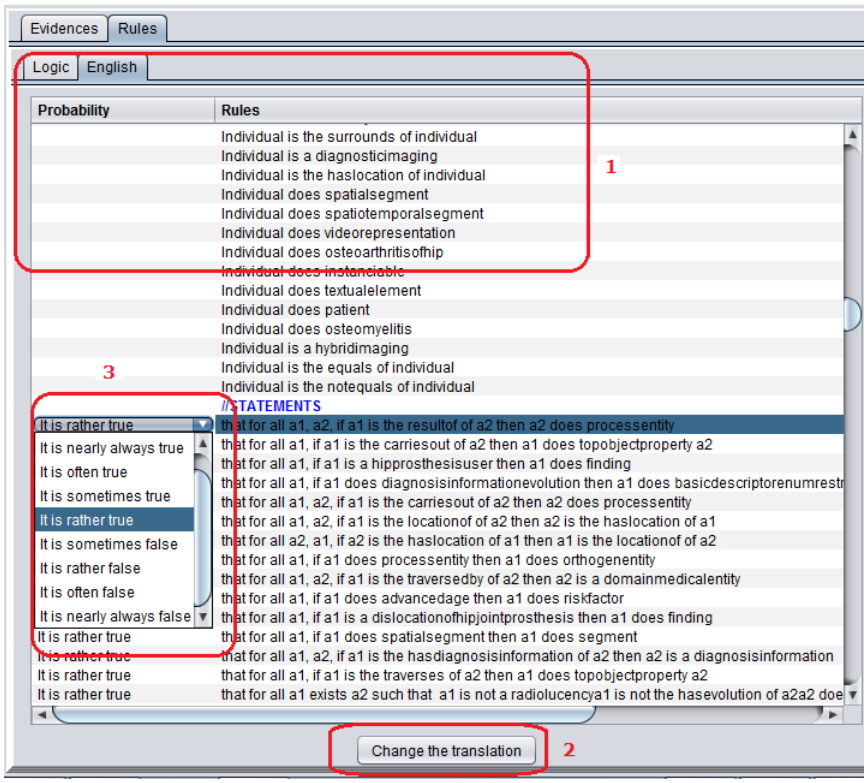
## 3.7 De la logique du premier ordre vers le langage naturel

Afin de faciliter la communication entre les experts médicaux et les membres du projet ORTHOGEN, nous devions implémenter un processus permettant de transformer des règles écrites en logique en premier ordre vers l'anglais.

La raison d'être de ce processus provient du fait que notre programme pourrait être utilisé par deux profils de personnes.

Tout d'abord, le premier type serait un ingénieur de connaissances, spécialiste en logique du premier ordre, qui ne rencontrerait dès lors aucune difficulté dans la compréhension de la logique. Dans le cadre de notre travail, il pourrait s'agir d'un membre de l'équipe ORTHOGEN, mais comme nous l'avons déjà exprimé, si l'on sort du contexte de notre projet, il pourrait s'agir de toute personne ayant des compétences en informatique et en logique.

FIGURE 3.9 – Module de traduction




Le second type de profils serait un professionnel en médecine qui aurait besoin de vérifier la véracité d'une règle selon sa propre expérience. N'ayant pas de familiarité avec la logique du premier ordre, il est alors nécessaire de lui proposer ces règles dans un langage qu'il connaisse. C'est pour cela que nous avons choisi l'anglais. Ce type de profil correspond aux médecins de Mont-Godinne dans le cadre du projet ORTHOGEN et pourrait s'étendre à tout professionnel d'un domaine non informatique si l'on sort de ce cadre.

Pour cette raison et afin d'améliorer la communication, nous avons introduit au sein de notre programme une partie traduction qui permet en naviguant d'un onglet à un autre de passer de la logique du premier ordre au langage naturel comme il est représenté dans la partie 1 de la figure 3.9. Ce processus traduit aussi bien la partie des évidences ou des déclarations que la partie des règles par des processus que nous expliquerons tout au long de cette section.

Hormis la traduction de ces règles logiques, nous avons également traduit les poids selon des intervalles de valeurs. Ainsi, les règles sont introduites par

des propositions allant de « Il est parfois vrai que » pour les règles ayant les plus petits poids jusqu'aux des règles toujours vérifiées introduites par la proposition « Il est toujours vrai que ». Ces expressions existent également dans le cas de règles comportant des poids négatifs, ce qui se répercute par une négation des propositions ci-dessus. Lors du dialogue avec un médecin, il peut s'avérer que ce dernier ne soit pas d'accord avec la proposition qui accompagne la règle ; il pourrait, par exemple, déclarer qu'une règle ne se produit non pas « parfois », mais bien dans tous les cas, ce qui signifierait que l'utilisateur de notre outil devrait modifier la proposition accompagnant cette règle. Dans de tels cas, il suffit à ce professionnel de sélectionner cette proposition parmi une liste prédéfinie comme nous le voyons dans à la partie 3 de la figure 3.9 et cela aura pour répercussion implicite de changer le poids attaché à cette règle dans la partie logique.

Concernant l'implémentation de ce processus, nous avons dans un premier temps effectué une certaine recherche afin de vérifier si ce travail n'avait pas déjà été réalisé par d'autres personnes. Nous avons retenu deux articles décrivant le passage de la logique du premier ordre à l'Anglais. Ces deux-ci ne présentaient pas la même méthode pour arriver au résultat final et aucune d'elle ne nous convenait entièrement. Nous avons donc pris des connaissances dans chacune d'elles afin de les adapter à notre propre solution.

Tout d'abord, ces deux documents préconisaient l'utilisation de Jess[6], un langage de programmation orienté règles, car celui-ci était très utile afin de raisonner sur la structure des règles en logique du premier ordre. Cependant, nous avons fait le choix de ne pas utiliser cet outil. Différentes raisons ont poussé notre choix.

Premièrement, les documents que nous avons consultés partent de l'implémentation de Jess en montrant que ce dernier utilise sa grammaire spécifique. Cependant, dans notre cas, nous possédions déjà notre propre grammaire. Nous avons ainsi préféré partir de celle-ci plutôt que de la transformer en un format compréhensible par Jess.

De plus, cette dernière démarche, en plus de provoquer une lourde charge de travail supplémentaire, aurait également fait disparaître l'avantage que nous possédions grâce à l'exécution de notre grammaire comme décrit dans la section 3.5. En effet, nous avions déjà une implémentation d'un arbre syntaxique en Java. Il nous suffisait d'appeler une méthode déjà existante pour récupérer le contenu de cet arbre. A partir de ce point, il nous était facile de créer de nouvelles méthodes partant de cet arbre et d'appliquer les règles de traduction décrites dans cette section.

6. http ://www.jessrules.com/

Pour les deux raisons que nous venons d'évoquer ci-dessus, nous n'utiliserons pas Jess mais garderons notre propre grammaire pour laquelle nous créerons simplement une nouvelle méthode.

Concernant la réalisation de cette méthode, nous avons dans un premier temps opté pour une implémentation plus proche de celle que propose le premier article d'Aikaterini Mpagouli et d'Ioannis Hatzilygeroudis [33](Aikaterini Mpagouli et Ioannis Hatzilygeroudis). En effet, celui-ci propose de transformer toutes les variables contenues dans les prédicats par la traduction du nom de ces prédicats. Ainsi, par exemple, la règle « $\forall x \ : human(x) \Rightarrow lives(x, Patras)$ » se traduira par « Every human lives in Patras » et ne contiendra dès lors plus aucune variable.

Nous avons tout d'abord pensé que ces traductions permettraient plus facilement à un professionnel en médecine de comprendre les règles car nous y supprimions toutes les variables qui pourraient venir polluer la traduction de celle-ci. Cependant, nous avons rencontré plusieurs problèmes. D'une part, au niveau du développement. En effet, lorsque nous rencontrions des cas limites, nous n'obtenions pas toujours les traductions les plus adéquates à cause de la difficulté à implémenter toutes les règles de traduction expliquées dans le document sur lequel nous nous basions. D'autre part, pour ces mêmes cas limites, les traductions proposées n'étaient pas toujours des plus compréhensibles. Par exemple, pour la règle suivante : « $\forall x \neg \exists y \forall z \ : (person(x) \wedge person(y) \wedge person(z) \wedge \neg respects(x, y)) \Rightarrow \neg hires(z, x)$ ». La traduction proposée était la suivante : « Every person that does not respect no person is not hired by any person ». Le surplus de négations rend la phrase très difficile à comprendre et augmente considérablement les chances de se tromper lors de la traduction de cette dernière.

Pour les deux raisons évoquées plus haut, nous avons commencé à nous intéresser plus attentivement au deuxième article d'Aikaterini Mpagouli et d'Ioannis Hatzilygeroudis [34](Aikaterini Mpagouli et Ioannis Hatzilygeroudis). Dans celui-ci, les variables étaient préservées lors de la traduction.

Au début, nous avons pensé que maintenir les variables dans la traduction pourrait handicaper la lecture de l'utilisateur mais après avoir observé un grand échantillon de traductions, nous sommes arrivés à la conclusion que l'utilisateur parviendrait sans problème à associer la variable à un individu ayant la caractéristique de son prédicat. Ainsi, la traduction « $x$ is a student » provenant du prédicat $student(x)$ est facilement compréhensible par tout type d'utilisateur malgré le fait que la variable ne soit pas éliminée.

Nous avons alors réalisé la structure de nos traductions à partir de ce second article d'Aikaterini Mpagouli et d'Ioannis Hatzilygeroudis [34](Aikaterini

Mpagouli et Ioannis Hatzilygeroudis). Cependant, bien que nous ayons abandonné l'idée de suivre son premier article, nous y avons tout de même gardé certaines particularités. Dans une conjonction de prédicats, nous attachons tous les prédicats associés à la même variable ensemble. Ainsi, la règle « $student(x) \wedge strongInMath(x) \wedge oftenStudies(x)$ » ne se traduira pas comme « $x$ is a student and $x$ is strong in math and $x$ often studies » mais plutôt par « $x$ is a student that is strong in math and does often study ».

Dans ce dernier exemple, nous pouvons par ailleurs remarquer que nous gardons également du premier article les différentes fonctions que peuvent prendre les prédicats dans la langue anglaise. Ceux-ci sont au nombre de trois : Il peut s'agir d'un nom, d'un verbe ou encore d'un adjectif.

Si l'on sait à laquelle de ces trois classes se rapporte chaque prédicat, nous pouvons apporter une traduction beaucoup plus précise à l'utilisateur. Cependant, un problème est vite apparu lors de notre effort à respecter cette idée. Il n'était pas facilement possible de déduire cette classe à partir du nom du prédicat qui pourrait même ne pas signifier grand-chose si l'utilisateur ne respecte pas les conventions de nommage. Nous avons ainsi créé une nouvelle option pour l'utilisateur : La possibilité, d'une part, de donner une traduction anglaise à son prédicat et, d'autre part, la possibilité d'attacher une fonction grammaticale à ce dernier. Cette option est proposée à la création de chaque prédicat, mais peut également être modifiée ultérieurement en sélectionnant un prédicat faisant partie des déclarations et en appuyant ensuite sur le bouton permettant de changer la traduction comme représenté dans la partie 2 de la figure 3.9. Ainsi, si l'utilisateur traite un prédicat non explicite tel que « $StuUnamur(x)$ », il pourrait spécifier que ce dernier se traduise par « student at Unamur » et qu'il s'agisse d'un nom. La traduction résultante serait alors « $x$ is a student at Unamur », ce qui ne poserait plus de problème de compréhension.

En plus de la possibilité qu'a l'utilisateur de spécifier cette fonction au sein de notre outil, nous lui avons également permis d'intégrer cette démarche à l'intérieur même du code de l'ontologie. En effet, notre programme, lorsqu'il va transformer une ontologie en règles de la logique du premier ordre, va également parcourir les commentaires de l'ontologie à la recherche de délimiteurs propres à la traduction anglaise. Il en existe deux. Le premier peut être représentés par "Type :Noun" pour spécifier la classe, où « Noun » peut être changé en « Verb » ou encore « Adjective » suivant la fonction de ce prédicat. Le second quant à lui sera de la forme "PredicateValue : « nom du prédicat » " et permettra d'indiquer le nom exact par lequel il faudra traduire ce prédicat.

Bien que notre traduction ait été fortement améliorée grâce à ce dernier point, le problème des clauses conditionnelles persistaient. En effet, celles-ci se divisaient en deux parties distinctes.

Tout d'abord, les règles comportant de multiples clauses conditionnelles. Celles-ci devenaient très difficiles à traduire et entrainaient des résultats non compréhensibles comme ceux déjà évoqués précédemment.

Ensuite, les règles comportant une seule clause conditionnelle. Pour celles-ci, il nous était nécessaire de traiter différemment les hypothèses de leur conséquence.

Pour pallier ces problèmes, nous avons tout d'abord tenté d'appliquer une traduction plus proche de l'anglais en nous basant sur le premier article de Mpagouli, notamment en insérant différents mots clé comme « that » dans les hypothèses afin d'introduire des phrases plus élégantes. Cependant, nous avons vite constaté que cette méthode engendrait de nouveaux cas limites très difficiles à traiter. Nous avons alors opté pour une solution plus simple, mais qui ne perd toutefois pas de précision. Celle-ci consiste dans un premier temps à transformer les différentes implications contenues dans la règle en une seule en utilisant de simples règles d'équivalences de la logique du premier ordre. Ceci est réalisé au moment où l'on parcout l'arbre syntaxique. Une fois cette étape réalisée, nous avons alors séparé les hypothèses de la conséquence de la règle obtenue par la structure habituelle du « if – then ». Cette structure, bien que restant plus proche de la logique du premier ordre que d'une phrase élégante en langage naturelle, ne handicape nullement la lecture de la règle et ce même pour une personne totalement extérieure aux notions de logique.

Pour conclure cette partie, nous pouvons affirmer que la manière dont nous avons transformé toutes nos règles n'est pas la plus efficace au niveau de l'élégance. Cependant, ce processus permet de combiner des parties de la logique du premier ordre – en gardant les variables et la plupart des opérateurs – et la langue anglaise de la manière la plus simple possible. Cette simplicité n'entraine cependant aucune difficulté de compréhension et cela même pour des phrases longues et complexes. C'est pour cette raison que nous n'avons pas choisi de pousser plus loin cette partie traduction de notre outil.

## 3.8 Historique

Une fonctionnalité qui, bien que non essentielle pour l'objectif lié à notre outil, est importante pour l'utilisabilité de ce dernier est la fonction d'historique. Celle-ci est divisée en deux sous parties. D'une part, une partie « historique » permettant de naviguer dans les opérations réalisées sur un ensemble de règles grâce à des fonctions « Précédent » - « Suivant » et cela même après que notre programme ait été fermé. D'autre part, une autre partie « historique » permettant cette fois d'aider l'utilisateur lorsqu'il souhaite réappliquer des opérations qu'il avait déjà réalisées sur les règles provenant d'une ontologie lorsque cette même ontologie est transformée à nouveau après avoir subi des modifications.

Ces deux sous parties utilisent le même procédé, mais possèdent des particularités en fonction de leur objectif spécifique. L'idée derrière ce système d'historique est d'enregistrer toutes les modifications concernant un ensemble de règles dans un fichier pour ensuite pouvoir y naviguer au besoin. Ce fichier va contenir une liste de changements caractérisés par différentes propriétés que nous expliquerons plus loin. Cette liste de changements va ensuite être sérialisée à l'intérieur de ce fichier afin de pouvoir garder la structure de l'objet manipulable tout en facilitant les opérations de lecture et d'écriture sur le fichier. L'objet utilisé pour représenter un changement va comporter un certain nombre de propriétés.

Tout d'abord, un nombre entier correspondant au type du changement effectué. Les différents types possibles sont l'ajout, la suppression, la modification d'une ou plusieurs règles et l'apprentissage selon le raisonneur sélectionné.

Ensuite, un changement peut comporter deux autres propriétés représentées par deux listes de règles. Celles-ci peuvent être toutes les deux utilisées dans le cas d'une modification de règles. Dans ce cas-ci, la première liste contiendra la liste de règles à substituer par celles que contient la seconde. Il se peut également qu'une seule de ces deux listes soit utilisée, notamment dans le cas d'un ajout de règles. Seules la première liste va comporter les règles à ajouter à l'ensemble.

Grâce à ces trois propriétés, nous pouvons maintenant représenter toutes les possibilités concernant un changement précis, ainsi que les règles impliquées dans le changement. Nous pouvons ainsi en parcourant la liste de changements récupérée dans le fichier sérialisé avancer ou reculer parmi les opérations déjà effectuées comme le fait un système d'historique habituel.

Cependant, cette façon de faire engendre des complications dans le cas où l'on doit passer d'un outil à un autre. En effet, les différences de syntaxe nous posent des problèmes quant à la compatibilité des anciennes règles envers les nouvelles. Pour pallier ce problème, nous avons décidé que lorsque l'utilisateur décide de changer d'outil, nous transformerions l'entièreté des règles contenues dans l'historique de l'ancienne vers la nouvelle syntaxe et ce en utilisant le mécanisme de changement de syntaxe décrit dans la section 3.5. Cette opération ne se caractérise non pas par une modification de toutes les règles mais plutôt par une suppression des anciennes pour faire place à l'ajout de toutes les nouvelles règles présentes cette fois dans la bonne syntaxe. L'idée est d'ailleurs la même dans le cas d'un apprentissage. Etant donné que nous ne pourrions deviner l'ensemble des corrélations entre les règles précédant cette opération et celles qui en résultent, nous supprimons à nouveau les anciennes pour faire place aux nouvelles règles.

Ainsi, la puissance qu'offre ce système d'historique est telle que nous pouvons commencer par transformer une ontologie et obtenir les règles résultantes, ajou-

ter des règles à l'ensemble déjà existant, changer d'outil, effectuer un apprentissage, fermer le fichier et, par la suite, réobtenir les modifications effectuées au tout début de cette suite d'opérations en utilisant ce système après avoir ouvert à nouveau le fichier.

Cependant, un autre problème apparait lorsque l'on évoque ce scénario. Il s'agit du cas dans lequel nous devrions rejouer l'ensemble de ces modifications sur une ontologie qui aurait été améliorée. En effet, le système d'historique tel qu'il est actuellement décrit ne permet pas de retransformer une ontologie et d'obtenir les opérations précédemment effectuées dessus.

Nous avons ainsi mis en place ce nouveau système en nous appuyant sur le procédé déjà existant. L'idée que nous avons exploité est le fait que si un utilisateur souhaite créer un fichier dont le nom existe déjà, nous lui proposons – dans le cas où ce remplacement de fichier correspondrait à une évolution d'ontologie – de réappliquer toutes les opérations ayant été effectuées lors de la dernière version de l'ontologie. Ainsi, si cet utilisateur avait passé un long moment à réaliser 150 modifications, par exemple, il nous suffirait de consulter le fichier répertoriant la liste des changements de l'ancien fichier afin de réappliquer tous ceux-ci de manière automatique sur le nouveau fichier. Le cas échéant, l'utilisateur économiserait énormément de temps à ne pas devoir rejouer des opérations qu'il avait déjà effectuées dans le passé.

En outre, nous permettons également à l'utilisateur d'obtenir les changements qu'il avait effectués dans les traductions de ses règles. L'idée est la même que celle précédemment décrite. S'il avait passé un temps important à peaufiner toutes ses traductions, il serait dommage qu'il doive tout recommencer lorsqu'il transforme une ontologie modifiée.

Ainsi, grâce à notre double système d'historique, l'utilisateur a beaucoup moins de soucis à se faire quant aux opérations redondantes qu'il aurait déjà réalisées. Notre objectif était de fournir un procédé permettant à ce dernier un gain de temps et une utilisation plus souple de notre outil.

## 3.9 Statistiques

La dernière fonctionnalité que nous allons vous présenter concerne les statistiques. Un des buts principaux de notre outil étant de comparer les performances d'Alchemy, de Tuffy et de RockIt, il nous était nécessaire d'offrir à l'utilisateur des possibilités de comparaisons quant aux résultats obtenus.

Toute la gestion des statistiques repose sur un fichier XML contenant tous les fichiers temps et résultats obtenus par les différents raisonneurs de la logique de Markov.

Ce fichier est composé de plusieurs parties.

Tout d'abord une première partie reprenant les informations des différents fichiers. Celle-ci est composée du nom du fichier, du chemin menant à l'ontologie sur l'ordinateur de l'utilisateur du programme et enfin de la sélection des branches de l'ontologie qui seront utilisées pour la génération de règles. Celle-ci peut contenir l'entièreté de l'ontologie comme une ou plusieurs sous-parties de cette dernière comme nous l'avons évoqué dans la section 3.4.

Ensuite, la deuxième partie se rapporte aux différents temps d'exécution. Celle-ci contient tout d'abord le nom du fichier permettant de l'associer à un élément de la première partie. Elle contient également un indicateur indiquant si le ou les temps concerne la phase d'apprentissage ou d'inférence. Ensuite, elle contient une liste de requêtes sur lesquelles a été réalisé l'apprentissage ou l'inférence. Enfin, elle contient les temps pour les raisonneurs ayant réalisé cette requête précise. Cette structure est intéressante car elle permet facilement de comparer les temps entre les différents outils ayant effectué une requête semblable.

Enfin, la dernière partie comporte les résultats retournés par l'inférence des différents outils. La composition de cette partie commence bien sûr par le nom du fichier lié à un élément de la première partie. A nouveau, cet indicateur nous permet de réaliser des liens entre les différentes parties du fichier XML. Une subtilité de notre structure réside dans le fait que chaque résultat ne comporte pas la liste des éléments retournés par outil ou par fichier, mais bien un prédicat associé à un fichier. Par prédicat, nous entendons, un élément retourné par un raisonneur. Celui-ci est instancié par une constante existant dans la liste des évidences et se voit associer une probabilité. Un exemple de prédicat comme nous l'entendons pourrait être $Student(Joseph)$ auquel va être attaché une probabilité de 0.5 qui sera différent de Student($Mathieu$) qui, lui, représenterait un autre prédicat. En outre, pour chaque prédicat associé à un fichier en particulier, nous indiquons par quelle requête ce dernier a été obtenu. Nous exprimons, enfin, les probabilités renvoyées par les différents outils qui possèdent ce résultat. A nouveau, ce procédé nous permet très facilement de comparer les différentes probabilités suivant l'un ou l'autre outil.

Maintenant que nous avons la structure du fichier XML, il nous faut trouver un moyen de peupler celui-ci lors des différentes opérations de l'utilisateur. Ce processus de peuplement va être réalisé à chaque fois que l'on effectue un apprentissage ou une inférence sur un ensemble de règles. Cette démarche va être divisée en trois sous processus.

Tout d'abord, si c'est la première fois que l'on réalise un apprentissage ou une inférence sur le fichier content l'ensemble de règles, un nœud va être créé dans

la première partie reprenant les différents éléments constituant la structure du fichier XML évoqué ci-dessus.

Ensuite, en ce qui concerne le processus de comparaison de temps, ce dernier, après avoir récupéré les différents paramètres dont aura besoin le fichier XML, va calculer le temps d'exécution du processus d'apprentissage ou d'inférence. Si la requête en question n'avait pas encore été effectuée sur le fichier contenant l'ensemble de règles, le programme va alors créer une nouvelle sous-section dans la structure XML et va y ajouter ce premier temps. Sinon, le programme va parcourir cette dernière afin de récupérer le nœud correspondant à cette sous-section, puis va simplement y ajouter le nouveau temps calculé avec le raisonneur de la logique de Markov sélectionné.

Enfin, concernant le processus de comparaison des résultats, ce dernier est assez similaire. La différence réside dans le fait que le processus de récupération d'un nœud déjà existant ou de création d'un nouveau ne va pas se faire sur l'entièreté de la requête, mais sur chaque prédicat retourné en résultat de l'inférence.

Maintenant que notre processus de stockage et de peuplement est opérationnel, nous pouvons proposer cette fonctionnalité à l'utilisateur. Celle-ci va se présenter dans notre programme par un onglet « Statistics » offrant deux possibilités. La première permettant de comparer des résultats entre les différents outils. La seconde quant à elle permettant de comparer cette fois-ci les temps de ces outils.

Les figures 3.10 et 3.11 montrent un aperçu de l'utilisation de ces deux fonctionnalités.

Lorsque la première option est sélectionnée, une boîte de dialogue – reprise à la figure A.9 – propose d'afficher uniquement les résultats dont la différence de probabilité entre les outils atteint un certain seuil. Par exemple, si les résultats retournés par Alchemy, Tuffy et RockIt pour un prédicat donné sont les suivants « 0.2, 0.35, 0.55 » et sont « 0.2, 0.25, 0.3 » pour un second prédicat. Si une différence minimale de 0.3 est choisie, seul le premier résultat sera affiché par notre programme. Cela permet à l'utilisateur de se focaliser sur les plus gros écarts. En effet, s'il souhaite chercher des résultats très différents afin de comprendre la raison de cet écart, il ne souhaite pas être encombré par une grosse panoplie de résultats plus cohérents.

La seconde option de ces statistiques permet de comparer les temps entre les différents outils pour un fichier et une requête précise. Cette option permet simplement à l'utilisateur de sélectionner le fichier qu'il souhaite parmi la liste de tous ceux existant dans le fichier XML. Il peut ensuite choisir de comparer les temps d'apprentissage ou d'inférence et obtient la comparaison des résultats

FIGURE 3.10 – Statistiques - Comparaison de résultats

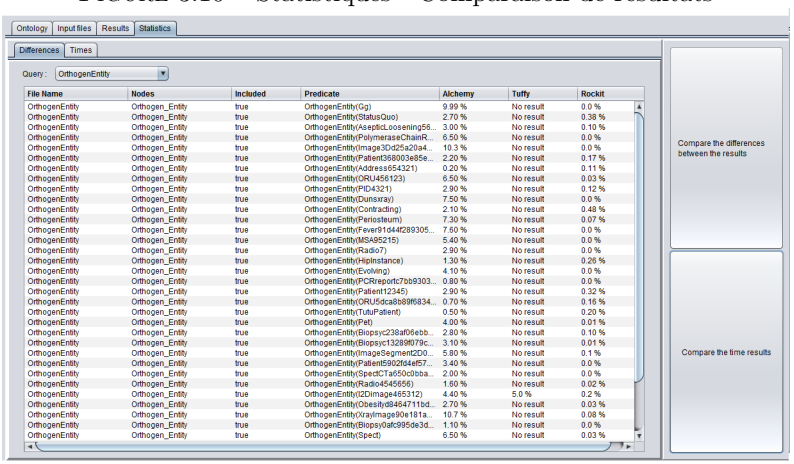
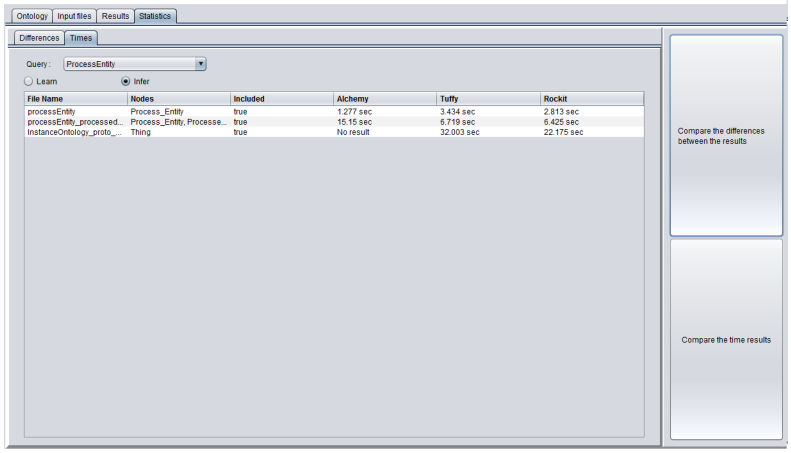



FIGURE 3.11 – Statistiques - Comparaison de temps



souhaités. Il peut ainsi voir et analyser en parcourant plusieurs cas quel outil semble être le plus rapide dans quel cas.

En combinant ces deux possibilités de statistiques, l'utilisateur a accès à un grand nombre d'information quant à la pertinence des résultats d'une part et à la rapidité des outils d'autre part. Les résultats obtenus plus loin dans ce mémoire se sont d'ailleurs basés sur cette aide afin d'obtenir une relation entre la pertinence et la rapidité de chaque outil.

# Chapitre 4

# Expérimentation

**Sommaire**



## 4.1 Introduction

Une fois notre phase de développement terminée et notre programme fonctionnel, nous avions en mains tous les outils nécessaires pour réaliser nos différents cas de tests. Après la mise en place et l'analyse de ces derniers, il nous était alors possible de tirer les conclusions que nous souhaitions quant à l'efficacité des trois moteurs d'inférence probabilistes que nous avions sélectionnés.

Comme nous l'avons déjà évoqué dans la partie concernant les objectifs de ce mémoire, cette phase d'expérimentation est primordiale pour le projet ORTHOGEN car elle va permettre de connaître les avantages et les désavantages de chacun de ces raisonneurs de la logique de Markov et, par conséquent, elle permettra également de savoir dans quel contexte l'un ou l'autre outil devra être utilisé. Ce processus est primordial car le programme final devra être utilisé dans le domaine médical et nécessitera alors le plus haut degré de précision qu'il est possible d'offrir grâce à ces trois moteurs d'inférence probabilistes.

Pour commencer notre phase de tests, nous avons utilisé l'ontologie réalisée par le projet ORTHOGEN dont le domaine d'application reprend les infections de prothèses orthopédiques. Ce premier choix est évident car le projet final doit

être amené à travailler en particulier sur cette ontologie. Il parait alors tout à fait normal que les conclusions que nous tirerons soient applicables à cette dernière. Concernant sa complexité et sa conséquence en terme de règles, celle-ci s'avère être un cas d'étude intéressant. Elle est composée de 84 évidences et de 338 clauses logiques, ce qui fait de cette ontologie une structure assez importante en ce qui concerne son nombre de règles. Il est alors certain que tester cette ontologie nous permet de retirer des résultats à la fois riches et intéressants.

Cependant, cette dernière, seule, ne reprend pas un nombre élevé de cas de tests. Nous avons alors, par la suite, testé des sous partie du domaine de connaissances de cette ontologie afin d'obtenir des domaines qui devenaient graduellement de plus en plus grands. La réalisation de cette tâche nous a été facilitée grâce à deux facteurs existants. Tout d'abord, grâce à la fonctionnalité de notre programme qui permettait de choisir de ne transformer qu'une sous partie du domaine de connaissances d'une ontologie en règles de la logique du premier ordre. Ensuite, le deuxième élément qui a facilité la réalisation de cette tâche est la richesse de l'ontologie d'ORTHOGEN en termes d'héritage. En effet, lorsque nous ouvrions cette dernière avec un éditeur d'ontologie comme, par exemple, Protege, nous pouvions constater que celle-ci comportait plus d'une centaine de classes et de ramifications de ces classes en sous classes. Ainsi, il était très simple de choisir de manière progressive un sous ensemble du domaine de connaissances de plus en plus important jusqu'à arriver à l'entièreté de l'ontologie elle-même.

Après avoir élaboré une série de tests concernant des ramifications de l'ontologie ORTHOGEN, nous avions couvert un nombre plus important de cas de tests. Cependant, il nous fallait encore continuer à tester d'autres voies afin de couvrir des cas plus limites que la première ontologie n'a pas pu tester.

Nous avons alors choisi de réaliser de nouveaux tests sur d'autres ontologies. Premièrement, nous sommes restés dans le domaine médical bien que la matière traitée ne concerne plus de l'orthopédie. Ensuite, nous nous sommes également dirigés vers d'autres domaines d'application pour réaliser nos tests. La raison de ce choix est double. D'une part, comme nous l'avons évoqué, notre outil nous permet de traiter n'importe quel domaine sans que cela n'impacte son fonctionnement ou les résultats obtenus. D'autre part, les résultats que nous obtenons grâce aux raisonneurs de la logique de Markov se doivent d'être corrects indépendamment du domaine dans lequel ils ont été induits. Se tourner vers d'autres disciplines pouvait alors nous apporter plus de cas limites et ainsi enrichir nos résultats et par conséquent nos conclusions.

Au total, nous avons tenté de couvrir le plus de cas différents. Nous avions bien sûr des ontologies comportant des domaines de connaissances importants alors que d'autres étaient composées de plus petits domaines. Mais d'autre part, nous avons veillé également à ce que les proportions de règles et d'évidences

au sein de chaque domaine varient. Ainsi, nous testions des domaines comportant beaucoup de règles avec beaucoup d'évidences. D'autres comportaient beaucoup moins d'évidences que de règles bien que ces premières ne soient pas négligeables. D'autres encore étaient composés d'un grand nombre de règles et de peu d'évidences. A l'inverse, nous avons également des domaines comportant un nombre très importants d'évidences pour très peu de règles. Au total, parmi les ontologies que nous avons testées, le nombre de règles varient entre 2 et 621 éléments et le nombre d'évidences varient entre 2 et 147 éléments. Les domaines variant parfois beaucoup les uns par rapport aux autres, nous espérions ainsi obtenir des résultats vrais dans tous les cas et non indépendants d'un domaine particulier.

Une fois l'ensemble des cas de tests réalisés, la dernière étape consistait en l'analyse des résultats obtenus. Le but de l'exercice était d'extirper de ces résultats des tendances vraies pour tous les scénarios et pouvant amener à des conclusions sur quel outil choisir dans tel ou tel cas. Concernant ces résultats obtenus, nous pouvions les diviser en quatre catégories.

Tout d'abord, il nous fallait différencier d'une part tous les tests réalisés sur l'apprentissage. Cette phase bien que très importante pourrait ne pas être utilisée de manière régulière. En effet, tant qu'un domaine de connaissance n'est pas modifié – que ce soit au niveau des règles ou des évidences – il n'est pas nécessaire de réaliser un nouvel apprentissage. Ensuite, il nous fallait comparer les résultats réalisés dans le cadre de l'inférence. Ceux-ci, en opposition à la première phase, sont plus prompts à être utilisés couramment car c'est cette phase d'inférence qui sera effectuée lorsque l'on souhaite obtenir des résultats concernant un patient en particulier.

Pour chacune de ces phases, il nous fallait alors comparer d'une part le nombre de résultats obtenus et d'autre part le temps qu'il a fallu pour obtenir ces résultats. Ces deux comparaisons vont permettre d'établir dans quels cas devront être utilisés les trois outils.

Concernant les comparaisons de temps, il est évident que ce facteur « temps » est très important dans le monde informatique. En effet, si nous remarquons qu'un outil propose des résultats dans des temps très courts et qu'il est viable pour un médecin d'attendre l'exécution totale du processus avant d'obtenir les résultats escomptés, nous pourrions utiliser cet outil dans un contexte de calcul en temps réel. S'il faut moins de 5 secondes à un processus pour une taille de domaine prédéfinie, nous savons qu'il n'est pas contraignant pour le médecin d'attendre la fin du diagnostic du programme. A l'inverse, si des résultats sont obtenus après un long moment d'attente qui peut aller de plusieurs heures à plusieurs jours, nous pourrions toujours envisager le processus si la pertinence et la précision des résultats sont importantes, mais cette fois-ci, le contexte du calcul en temps réel serait exclu. Il faudrait alors se tourner vers une solution

où le processus serait réalisé de manière périodique ou uniquement dans des cas où l'urgence n'est pas le mot d'ordre.

Concernant le nombre de résultats obtenus cette fois-ci, il est évident que ce facteur traite de la quantité mais pas forcément de la qualité. En effet, rien ne sert d'avoir un grand nombre de résultats si ceux-ci sont erronés. Cependant, ce facteur peut toutefois avoir une importance non négligeable. En effet, certains raisonneurs de la logique de Markov peuvent ne retourner qu'un petit nombre de résultats alors que d'autres en retourneraient un nombre beaucoup plus grand pour le même domaine et la même requête. En s'abstrayant de la qualité, il pourrait être plus intéressant dans certains contextes d'utiliser un outil qui retournerait une centaine de résultats plutôt qu'un autre qui n'en retournerait que deux. En effet, ce raisonnement peut se justifier par le fait que le premier permettrait au médecin d'obtenir une énumération des différents cas de maladies possibles en fonction des symptômes et cela, même si certains cas sont très peu probables, le second, quant à lui, ne renverrait que les deux cas les plus probables. Par conséquent, dans ce dernier cas, le médecin pourrait ne pas envisager un cas limite car il n'apparaîtrait pas dans les solutions. Le premier cas, bien que contenant du bruit – dans le sens où la plupart des cas ne se produiront presque jamais – aurait l'avantage de comporter des cas rares qui lorsque cela arriverait seraient affichés également au médecin.

C'est donc dans ce sens que nous avons réalisés les analyses de nos différents cas de tests. Nous vous exposerons les tendances et les résultats obtenus, ainsi que les conclusions qui en découlent dans la prochaine partie.

En outre, les tests ont tous été réalisés sur la même machine afin de ne pas biaiser les résultats obtenus en fonction des performances de l'ordinateur utilisé. La configuration de cette machine consistait en un processeur Intel core i7-2670 QM 2.2 GHz. La mémoire RAM possédée par l'ordinateur effectuant les tests était de 8GO. De plus, cette machine utilisait Windows 7 64 bits comme système d'exploitation.

## 4.2 Approche théorique

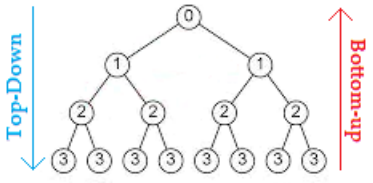
Dans cette section, nous allons passer en revue l'ensemble des techniques et technologies utilisées par les trois raisonneurs de Markov qui peuvent avoir une influence sure la complexité en temps ou encore l'espace mémoire utilisé par les deux processus principaux que nous allons tester par la suite qui sont l'apprentissage de poids et l'inférence. Nous allons analyser l'apport de l'utilisation de bases de données, l'implémentation d'une stratégie plutôt qu'une autre, ou encore les avantages d'un algorithme par rapport aux autres. Par ordre d'outils, nous allons commencer par Alchemy.

### 4.2.1 Alchemy

Alchemy est un raisonneur qui utilise la plupart de son temps d'exécution pour la phase d'instanciation du réseau [25](Niu et al.,2010). Il ne nécessite pas d'installation d'une base de données et donc l'ensemble des mécanismes se produisent en mémoire centrale. Lorsque l'on n'utilise pas de bases de données et que le domaine est assez large, il est probable que le raisonneur atteigne la limite de son espace mémoire et qu'il ne puisse pas terminer son processus ou effectuer son travail correctement [25](Niu et al.,2010). Alchemy est incapable de tirer profit d'un partitionnement de réseau logique de Markov pour pouvoir effectuer du parallélisme et gagner en performance. Alchemy possède l'implémentation de plusieurs algorithmes d'apprentissage et d'inférence [35](Kok, Singla, Richardson, Domingos et Summer,2007) mais nous avons choisi d'effectuer nos tests avec les plus performants d'entre-eux. L'algorithme MC-SAT pour l'inférence [32](Poon et Domingos) et l'algorithme de gradient conjugué avec préconditionnement pour l'apprentissage [30](Lowd et Domingos).

**Approche Top-Down** Lors de son instanciation de réseau, Alchemy utilise une stratégie « top-down » [25](Niu et al.,2010). L'idée d'une approche « top-down » est de démarrer avec une vue globale du problème, de considérer le problème dans son ensemble et d'ensuite décomposer et raffiner celui-ci en sous-problèmes plus détaillés. Les sous-problèmes obtenus peuvent être décomposés de manière répétitive jusqu'à atteindre un niveau de granularité non décomposable et une complexité plus abordable. Cette approche permet d'avoir une vue non détaillée du problème dans sa globalité très rapidement. C'est une stratégie qui s'oppose radicalement à la stratégie « bottom-up ». Le principe de cette dernière est de premièrement s'occuper des sous-problèmes simples et détaillés pour ensuite les composer ensemble pour arriver à s'adresser à un problème plus complexe. On compose les sous-problèmes jusqu'à atteindre le problème de départ. C'est donc une stratégie ascendante. La figure 4.1 donne une rapide idée des stratégies « top-down » et « bottom-up ». Dans le cas de l'instanciation au sein d'Alchemy, le processus démarre de la formule logique non instaniciée de départ, la décompose en clauses plus petites et instancie chaque variable avec chaque constante du domaine pour finalement obtenir l'ensemble des atomes instanciés comme l'algorithme général présenté à la page 33.

FIGURE 4.1 – Approche Top-Down et Bottom-Up



**Instanciation paresseuse** Cependant, Alchemy permet d'utiliser une méthode appelée « lazy grounding » ou encore l'instanciation paresseuse [25](Niu et al.,2010). Cette méthode permet d'appliquer une heuristique pour améliorer le processus d'instanciation de réseau de Markov et ce en plus de l'heuristique de réduction de réseau que l'on a vu dans la section sur l'inférence dans les réseaux logiques de Markov (page 39). L'idée de base de cette heuristique est de ne pas conserver en mémoire les parties du réseau de Markov qui ne seront pas ou quasi pas utilisées. Par exemple, on peut ignorer lors de la phase d'inférence les clauses qui seront tout le temps vraies étant donné l'ensemble d'évidences et ce peu importe la valeur des autres atomes [25](Niu et al.,2010). Pour ce faire, il faut définir deux nouvelles notions : les atomes instanciés actifs et les clauses instanciées actives. Un atome instancié est considéré comme actif quand il peut changer de valeur à un point quelconque du raisonnement alors qu'une clause instanciée est active quand elle peut être violée lorsque l'on change aucune ou plusieurs valeurs d'atomes actifs [25](Niu et al.,2010). On dit qu'une clause est violée lorsque elle a une valeur de vérité vraie et que son poids associé est négatif ou lorsque elle a une valeur de vérité fausse et que son poids est positif. Alchemy a pour but de garder en mémoire uniquement un petit sous-ensemble de clauses instanciés actives et d'en activer d'avantage si cela est nécessaire durant le mécanisme d'inférence. L'algorithme 3 montre le pseudo-code de l'instanciation paresseuse implémentée par Alchemy [36](Doan, Niu, Ré, Shavlik et Zhang,2011) où l'on voit que l'on calcule tout d'abord l'ensemble des clauses actives, c'est à dire, les clauses instanciées qui ne sont pas satisfaites par les évidences et ensuite on rend actifs les atomes qui se trouvent dans les clauses que l'on vient de calculer.

**Algorithm 3** Instanciation paresseuse

**Require:** Un réseau logique de Markov instancié
Un ensemble d'évidences
**Ensure:** $A$, les atomes instanciés actifs
$G$, les clauses instanciées actives
$G \Leftarrow$ clauses instanciées qui ne sont pas satisfaites par les évidences
$A \Leftarrow$ atomes contenus dans les clauses $\in G$

Le problème qui se pose avec ce type d'heuristiques pour l'instanciation est que les algorithmes d'inférence qui viennent s'exécuter ensuite requièrent une instanciation complète des clauses qui composent le réseau de Markov. Il faut donc adapter les algorithmes existant pour les rendre applicables à une telle heuristique [37](Poon, Domingos et Summer). L'avantage c'est que la méthode pour rendre ces algorithmes « lazy » est générale et peut être appliquée aux différents algorithmes d'inférence. La méthode consiste à remplacer dans l'algorithme de départ les instructions d'écriture et de lecture des atomes instanciés, ainsi que d'ajouter une étape d'initialisation, sinon le reste de l'algorithme reste le même [37](Poon et al.) :

– Lecture : Si l'atome instancié est en mémoire alors on lit la valeur comme l'algorithme de départ. Sinon on retourne la valeur par défaut.
– Écriture : Si l'atome instancié est en mémoire alors on met à jour la valeur comme l'algorithme de départ. Sinon on active l'atome instancié en lui allouant de la mémoire et en lui assignant la valeur sauf, si cette valeur est la valeur par défaut alors on ne fait rien. En cas d'activation, on alloue également de la mémoire pour les clauses qui sont activées par l'atome instancié.

Il est possible de raffiner encore plus la méthode d'adaptation d'algorithme [37](Poon, Domingos et Summer).

### 4.2.2 Tuffy

Tuffy est un raisonneur plus récent que Alchemy. Il implémente l'algorithme « diagonal Newton » [36](Doan et al.,2011) pour l'apprentissage de poids qui est une méthode très performante même si elle l'est un peu moins que la méthode « scaled conjugate gradient » implémentée dans Alchemy [30](Lowd et Domingos). En ce qui concerne l'inférence, Tuffy implémente l'algorithme MC-SAT qui est le plus performant à l'heure actuelle.

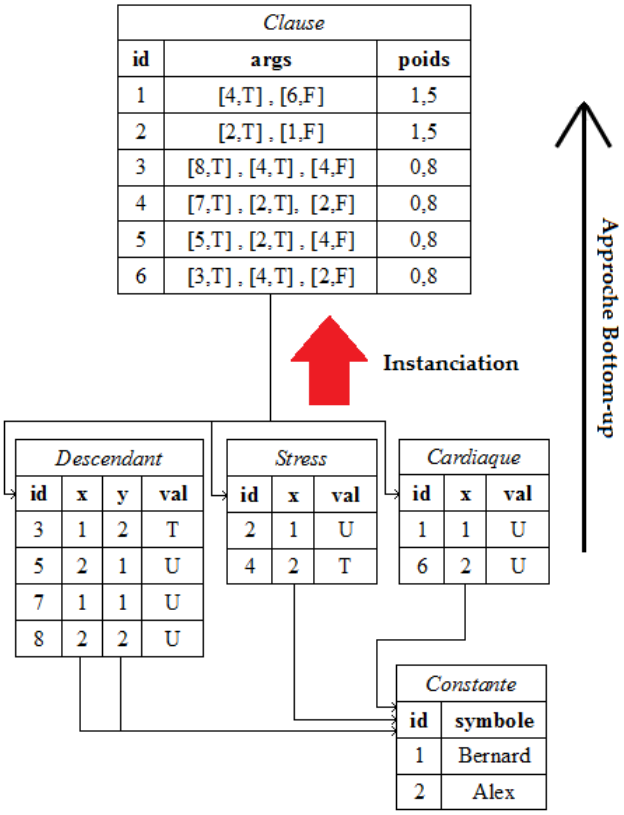
**Instanciation paresseuse** Il permet également d'exécuter l'heuristique d'instanciation paresseuse couplée à l'heuristique de réduction de réseau de Markov mais les développeurs de Tuffy se sont rendus compte qu'il était possible de répéter l'étape implémentée par Alchemy plusieurs fois jusqu'à la convergence des résultats et l'obtention de la fermeture de l'ensemble des clauses actives [25](Niu et al.,2010). On voit dans l'algorithme 4 que l'on répète l'étape de l'algorithme 3 jusqu'à ce que l'ensemble d'atomes instanciés ne grandisse plus [36](Doan et al.,2011).

**Algorithm 4** Instanciation paresseuse convergente

**Require:** Un réseau logique de Markov instancié
Un ensemble d'évidences
**Ensure:** $A_F$, les atomes instanciés actifs
$G_F$, les clauses instanciées actives
$A_F \Leftarrow \emptyset$
$G_F \Leftarrow \emptyset$
**repeat**
$G \Leftarrow$ clauses instanciées qui ne sont pas satisfaites par les évidences et les atomes inactifs ($\notin A_F$)
$A \Leftarrow$ atomes contenus dans les clauses $\in G$
$A_F \Leftarrow A_F \cup A$
$G_F \Leftarrow G_F \cup G$
**until** $A \subseteq A_F$

**RDBMS** Tuffy se démarque d'Alchemy par la mise en place d'un système de gestion de bases de données relationnelles (RDBMS) qui va permettre de changer de stratégie pour le processus d'instanciation et passer à une stratégie « bottom-up » [25](Niu et al.,2010). La base de données va permettre de ne plus conserver le réseau de Markov instancié en mémoire centrale et de ne plus avoir des problèmes de mise à l'échelle du domaine. Cela va permettre d'éviter les problèmes dû au dépassement de l'espace mémoire disponible. En effet, Tuffy permet de représenter un réseau instancié dans des tables contenues dans une base de données relationnelle [25](Niu et al.,2010). Si on reprend notre exemple de réseau logique de Markov, la figure 4.2 montre les différentes tables nécessaires et produites par le processus d'instanciation [36](Doan et al.,2011).

FIGURE 4.2 – Utilisation d'une base de données relationnelle

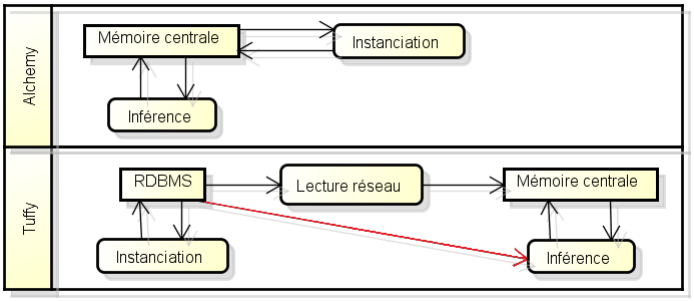



On voit dans cette figure que l'on démarre des atomes instanciés représentés par les tables « Cardiaque », « Stress » et « Descendant » et que après le mécanisme d'instanciation, on obtient la table qui représente les clauses instanciées. On est donc bien dans une stratégie « bottom-up » car l'on démarre

avec les éléments de base, les atomes, et on les compose ensuite pour finalement obtenir le réseau logique de Markov instancié. En entrée du processus d'instanciation, les tables représentant les atomes instanciés possèdent un identifiant qui est unique parmi l'ensemble des tables d'atomes, des références vers des constantes pour chaque variable dans l'atome et une valeur de vérité [25](Niu et al.,2010). La valeur de vérité peut être soit vraie(T) soit fausse(F) ou non spécifiée(U). L'instanciation produira alors la table « Clause » contenant l'ensemble des clauses instanciées. Cette table possède un identifiant, un ensemble d'arguments et un poids pour chaque clause [25](Niu et al.,2010). Un argument contient une référence vers un atome instancié et un booléen pour indiquer si oui ou non l'atome est nié dans la clause. Pour l'instanciation, Tuffy utilise des techniques standards de « bulk loading » [25](Niu et al.,2010). Ce sont des techniques qui sont désormais permises avec l'utilisation d'un RDBMS et qui vont permettre de charger de multiple lignes de données dans une table de la base de données. Cela permet donc de charger une large quantité de donnée de manière simultanée et rapide. Les optimisations apportées par le RDBMS permettent d'exploiter au mieux l'approche « bottom-up » et d'améliorer le temps nécessaire à l'instanciation [25](Niu et al.,2010).

**Architecture spécifique** Le RDBMS n'a malheureusement pas que des aspects positifs. En effet, les algorithmes d'inférence qui suivent la phase d'instanciation comme nous l'avons vu précédemment font de nombreux accès aléatoires aux données pour effectuer l'échantillonnage [25](Niu et al.,2010). Ils nécessitent de nombreux accès aux disques qui contiennent la base de données et dès lors, la complexité en temps de ces algorithmes est assez problématique. Il ne faut pas oublier qu'en plus l'exécution d'une étape de ces algorithmes d'échantillonnage va dépendre du résultat de l'étape qui précède. Dès lors, si ces algorithmes utilisent uniquement des accès aux disques, cela empêche d'exploiter de manière efficace le mécanisme de cache et le parallélisme que nous verrons plus loin [25](Niu et al.,2010). Pour aller à l'encontre de ce problème, Tuffy met en place une architecture particulière. A la figure 4.3, on peut voir l'architecture mise en place par Tuffy comparée à l'architecture du raisonneur Alchemy [25](Niu et al.,2010).

FIGURE 4.3 – Architecture Alchemy et Tuffy

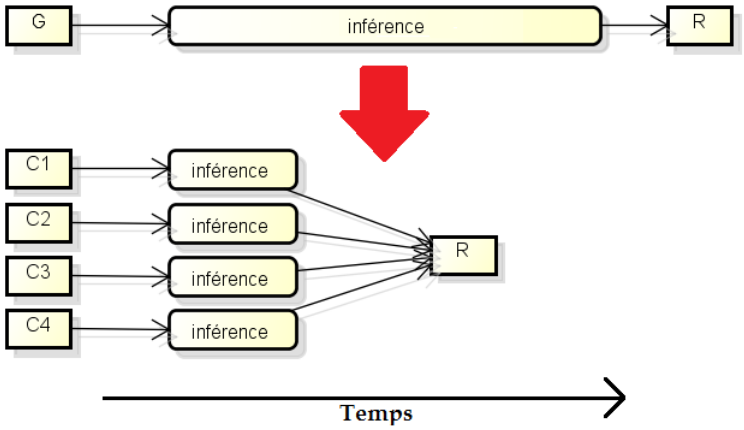


On voit dans la figure 4.3 que Tuffy permet de lire le réseau de Markov instancié depuis la base de données pour le stocker en mémoire centrale. Cela va permettre par la suite d'appliquer le processus d'inférence en mémoire centrale et ne plus avoir les inconvénients des nombreux accès aux disques [25](Niu et al.,2010). Cependant, vu que le RDBMS a essentiellement été mis en place pour pouvoir contrer les problèmes d'espace mémoire quand on doit traiter un domaine très large, il est possible que le réseau de Markov instancié soit trop large pour pouvoir être mémoriser en mémoire centrale. Dès lors, Tuffy permet également d'appliquer l'inférence directement sur la base de données dans le cas ou il ne serait plus possible de le faire en mémoire centrale [25](Niu et al.,2010). Ceci est représenté par la flèche rouge dans la figure 4.3.

**Partitionnement** Une autre amélioration que Tuffy apporte par rapport à Alchemy c'est la possibilité de partitionner le réseau obtenu après instanciation [25](Niu et al.,2010). On retrouve ici la volonté de diviser le problème en plus petites parties pour gagner en espace mémoire et en complexité au niveau du temps. L'idée derrière ce partitionnement est que l'on peut optimiser les probabilités des sous-parties du réseau de Markov et d'ensuite les composer pour obtenir l'optimisation de départ recherchée sans diminuer la qualité du résultat par rapport au résultat obtenu lors de l'application de l'algorithme d'inférence à l'entièreté du réseau [25](Niu et al.,2010). Pour ce faire, il faut parvenir à décomposer le réseau initial en sous-composants disjoints auxquels on appliquera l'algorithme d'inférence. Cela permet de pouvoir traiter des réseaux qui, d'une seule pièce, étaient trop larges pour la mémoire centrale. L'idée de séparer le réseau en composants est assez intuitive car la plupart du temps les règles contenues dans les réseaux logiques de Markov modélisent les interactions de ce qu'on peut appeler les entités du domaine modélisé et donc on peut faire correspondre un composant à la fermeture transitive d'une entité dans le graphe du réseau [25](Niu et al.,2010). Dans la figure 4.4, on peut voir le passage de l'aspect séquentiel de l'inférence à un aspect de parallélisme où $G$ représente le graphe de Markov non décomposé et $C_i$, le $\text{i}^{\text{ème}}$ composant de $G$.

Durant ce partitionnement, on essaye d'obtenir des composants de tailles plus ou moins identiques et de minimiser l'information perdue [25](Niu et al.,2010). Pour ce faire, il faut minimiser le poids total des clauses qui enjambent plusieurs partitions en réduisant ce nombre de clauses. Trouver le partitionnement qui minimise ces poids est un problème de complexité non polynomiale et l'algorithme va plutôt utiliser une heuristique plus abordable. Un des problèmes s'avère être que le nombre de composants obtenus lors du partitionnement peut devenir énorme ce qui rend non performant le chargement en mémoire de ceux-ci lors de l'inférence surtout quand le domaine est assez large [25](Niu et al.,2010). Il faut éviter de décomposer autant le réseau de Markov. Pour exploiter au maximum la décomposition en sous-parties sans perdre trop de temps dans le chargement de tous les composants, il est nécessaire de regrouper les composants en lots qui approchent le plus possible l'espace mémoire disponible [25](Niu et al.,2010). Ces

FIGURE 4.4 – Séquentialité vs parallélisme




lots ont pour seule contrainte de ne pas dépasser l'espace mémoire disponible. Cela relève de l'optimisation combinatoire qui est un problème NP-complet. Cependant, on peut appliquer une heuristique qui est un simple algorithme de tri appelé « first fit decreasing » qui ne permet pas d'atteindre l'optimum mais permet d'obtenir de bons résultats tout de même [25](Niu et al.,2010). Cet algorithme fonctionne en triant l'ensemble des composants dans l'ordre décroissant de leur taille et ensuite il parcourt cette liste et met le composant courant dans le premier lot qui est capable de le contenir. C'est à dire tant que le lot ne dépasse pas l'espace mémoire disponible pour l'inférence.

Il faut ensuite adapter les algorithmes de recherche sur lesquels sont basés les méthodes d'inférence pour qu'ils soient au courant du partitionnement. En effet, certaines clauses peuvent enjamber plusieurs partitions. C'est à dire qu'elle dépendent d'atomes se trouvant dans des partions différentes. Dès lors, il faut adapter les algorithmes de recherche pour qu'ils puissent gérer les dépendances entres les partitions quand cela est nécessaire [25](Niu et al.,2010).

**Parallélisme** Une fois que l'on a les différents lots contenant les composants du réseau, on peut tirer pleinement profit du mécanisme de parallélisme et exploiter l'entièreté des processeurs disponibles dans la machine virtuelle java. Le parallélisme nécessite juste de définir une politique d'ordonnancement entre les différents threads. Tuffy met en place la politique « round robin » [25](Niu et al.,2010) mais il serait intéressant de comparer les résultats avec une autre politique. La politique « round robin » se veut équitable et donne à l'ensemble des processus un temps de processeur équivalent et ce, chacun à leur tour. Quand un processus termine son tour de processeur, il est remit en bas de la liste, etc. Le parallélisme permis grâce au partitionnement permet de faire un gain de temps non négligeable [25](Niu et al.,2010).

Tuffy est un raisonneur qui va permettre de gérer les poids négatifs associés aux clauses. Pour ce faire, il va prétraiter les clauses avant qu'elles ne soient instanciées [38](Noessner,2013). Pour gérer ces poids négatifs, Tuffy transforme ces poids en poids positifs et nie les clauses qui y sont associées. Cependant, cette négation donne lieu à des conjonctions dans les formules et Tuffy est incapable de traiter les clauses contenant des conjonctions [38](Noessner,2013). Dès lors, il faut une nouvelle fois transformer les clauses pour n'obtenir que des clauses contenant des disjonctions. Ces deux manipulations de formules peuvent mener à des clauses qui ne sont plus du tout similaires aux clauses de départ et cela peut être considéré comme un désavantage de Tuffy [38](Noessner,2013) car les clauses ne respectent plus complètement la modélisation de départ. Cela peut avoir comme impact, non pas de fausser les mécanismes de calcul de probabilités, mais de fausser le modèle sur lequel ces probabilités sont calculées.

### 4.2.3 RockIt

RockIt est le raisonneur de la logique de Markov le plus récent des trois outils pris en compte. Il se base sur deux améliorations immédiates. Tout d'abord, lors de l'inférence, il assure (contrairement à Alchemy) qu'aucune contrainte « dure » ne soit violée dans les états trouvés [38](Noessner,2013). Une clause est soumise à une contrainte « dure » quand son poids est infini. Cela vient du fait qu'Alchemy va traiter les contraintes « dures » en remplaçant uniquement le poids de la clause par un poids de très grande valeur [38](Noessner,2013). Ensuite, RockIt va permettre de traiter les poids négatifs de manière correcte et ce, sans faire les manipulations que l'on a vu dans la section sur Tuffy [38](Noessner,2013). Par contre, ce raisonneur n'utilise pas l'algorithme MC-SAT [38](Noessner,2013) pour l'inférence mais bien un simple algorithme d'échantillonnage de Gibbs qui est moins performant [32](Poon et Domingos). Tout comme Tuffy, RockIt utilise une base de données pour effectuer et mémoriser l'instanciation du réseau de Markov.

**Formalisme ILP** En ce qui concerne l'inférence, RockIt met en place une idée un peu plus particulière. En effet, les développeurs de RockIt se sont rendus compte qu'il était possible de faire correspondre un problème d'inférence traditionnel à un problème d'optimisation avec des contraintes linéaires et une fonction linéaire objectif à optimiser, le tout formant un problème de programmation linéaire en nombres entiers (ILP) [39](Noessner, Niepert et Stuckenschmidt). L'idée est de, après instanciation, formuler les clauses instanciées dans la syntaxe ILP qui est plus compacte et de faire correspondre à chaque clause une contrainte linéaire. Dès lors, le principe d'inférence consiste à initialiser une première solution avec l'ensemble des évidences données, de sélectionner via de multiple jointures de tables les clauses se trouvant dans la base de données qui sont violées, de transformer ces clauses violées dans la syntaxe ILP, d'ensuite les rajouter dans la solution courante et de résoudre jusqu'à ce qu'il n'y ai plus de clauses violées dans celle-ci et que la fonction objectif soit optimisée

[39](Noessner et al.). On peut voir l'architecture du mécanisme [39](Noessner et al.) dans la figure 4.8 ci-dessous.

Regardons de plus près la syntaxe ILP et la formulation de contrainte. Pour chaque clause instanciée, il faut associer une variable binaire $x_i$ à chaque atome instancié $i$ se trouvant dans la clause [39](Noessner et al.). Ces variables valent 1 si leur atome instancié a une valeur de vérité vraie et 0 sinon. Les évidences peuvent donc être représentées par les contraintes linéaires suivantes : $x_i \geq 1$ si l'évidence correspondante à $x$ est vraie et $x_i \leq 0$ si l'évidence correspondante est fausse. On peut dès lors formuler les autres contraintes, soit $g$ une clause instanciée et $w_g$ le poids associé à $g$ [39](Noessner et al.) :

- si $w_g > 0 : \sum_{i \in L^+(g)} x_i + \sum_{i \in L^-(g)} (1 - x_i) \geq z_g$
- si $w_g < 0 : \sum_{i \in L^+(g)} x_i + \sum_{i \in L^-(g)} (1 - x_i) \leq (|L^+(g)| + |L^-(g)|) z_g$
- si $w_g = \infty : \sum_{i \in L^+(g)} x_i + \sum_{i \in L^-(g)} (1 - x_i) \geq 1$

où $L^+(g)$ représente l'ensemble des atomes instanciés non niés contenus dans $g$, $L^-(g)$ représente l'ensemble des atomes instanciés niés contenus dans $g$ et $z_g$ est une nouvelle variable binaire pour exprimer les contraintes sur les clauses. Une fois qu'on a formalisé l'ensemble des contraintes, il reste à déterminer la fonction objectif que le solveur doit optimiser. Cette fonction objectif est définie par [39](Noessner et al.) :

$$\max \sum_{g \in G} w_g z_g$$

Dans la figure 4.5, on peut voir un exemple de transformation de clauses instanciées en formalisme ILP [39](Noessner et al.).

FIGURE 4.5 – Transformation en formalisme ILP

| Poids | Clause instanciée | Contrainte ILP |
|---|---|---|
| 2 | $\neg x_1 \vee \neg x_2 \vee x_3 \vee x_4$ | $(1 - x_1) + (1 - x_2) + x_3 + x_4 \geq z_1$ |
| $-1.7$ | $x_1 \vee x_2 \vee \neg x_3$ | $x_1 + x_2 + (1 - x_3) \leq 3 * z_2$ |
| $\infty$ | $x_1 \vee x_2 \vee \neg x_3$ | $x_1 + x_2 + (1 - x_3) \geq 1$ |

**Agrégation de contraintes** Il est également possible grâce au formalisme ILP d'agréger les contraintes linéaires dans le but de réduire le nombre de variables et le nombre de contraintes, ce qui va permettre au solveur ILP de diminuer son temps global d'exécution [39](Noessner et al.). L'agrégation permet également de rendre le solveur ILP plus apte à utiliser les heuristiques de détection de symétrie. Ce sont des heuristiques qui permettent au solveur ILP d'exploiter les symétries au sein des contraintes pour améliorer ses performances [39](Noessner et al.). Si deux clauses $g_i$ et $g_j$ possèdent le même poids $w$ et

qu'elles sont de la forme $g_i = l_i \vee c$ et $g_j = l_j \vee c$ où $l_i$ et $l_j$ sont des littéraux quelconques, alors ces deux clauses sont agrégeables [39](Noessner et al.). On procède à l'agrégation de contraintes après la sélection des clauses violées, au moment où on les transforme dans le formalisme ILP. Les algorithmes servant à obtenir une agrégation optimale des contraintes s'avèrent être de complexité en temps trop élevée par rapport à la résolution ILP. Dès lors, RockIt met en place un algorithme d'estimation qui va approximer l'agrégation optimale [39](Noessner et al.). La figure 4.6 montre un ensemble de clauses agrégeables et d'autres clauses qui ne le sont pas.

FIGURE 4.6 – Exemple d'agrégation de contraintes

| g | $l_i$ | c | w |
|---|---|---|---|
| $g_1$ | $\neg x_1 \vee$ | $y_1 \vee \neg y_2$ | 0.7 |
| $g_2$ | $x_2 \vee$ | $y_1 \vee \neg y_2$ | 3 |
| $g_3$ | $\neg x_3 \vee$ | $y_1 \vee \neg y_2$ | 3 |
| $g_4$ | $x_4 \vee$ | $y_1 \vee y_2$ | 3 |

| $l_i$ | c | w |
|---|---|---|
| $\neg x_1 \vee$ | $y_1 \vee \neg y_2$ | 0.7 |
| $x_2 \vee$ | $y_1 \vee \neg y_2$ | 3 |
| $\neg x_3 \vee$ | | |
| $x_4 \vee$ | $y_1 \vee y_2$ | 3 |

Une fois qu'on a reperé les clauses agrégeables, la transformation en contraintes ILP peut être appliquée (Figure 4.7) [39](Noessner et al.).

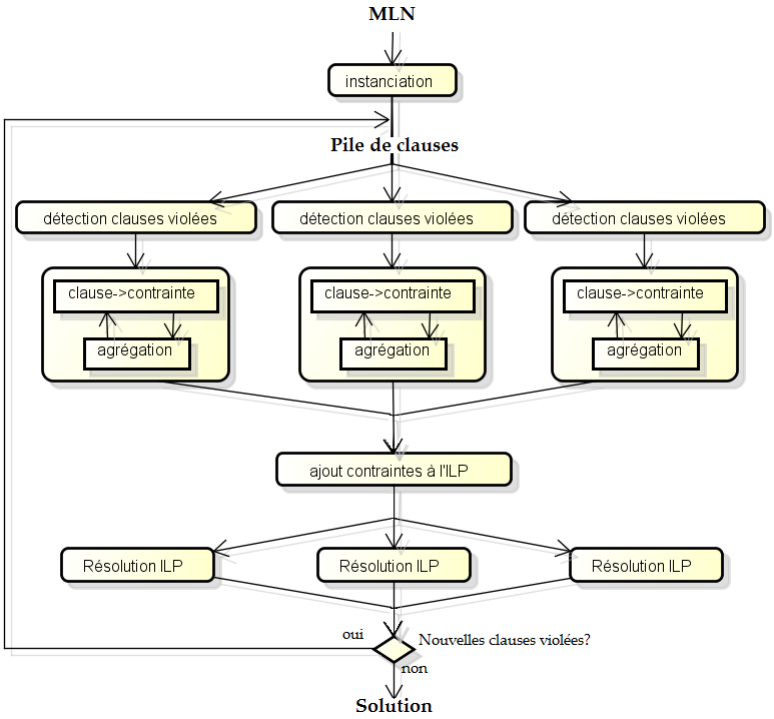
FIGURE 4.7 – Exemple de transformation en contraintes ILP de clauses agrégées

| g | $l_i$ | c | w |
|---|---|---|---|
| $g_2$ | $x_2 \vee$ | $y_1 \vee \neg y_2$ | 3 |
| $g_3$ | $\neg x_3 \vee$ | | |

| Contrainte ILP |
|---|
| $x_2 + (1 - x_3) \leq z_2$ |
| $2 \times y_1 \leq z_2$ |
| $2 \times (1 - y_2) \leq z_2$ |

**Parallélisme** Après avoir remarqué que bien souvent le temps de calcul des contraintes agrégées dominait le temps de la résolution ILP et pour tirer profit de la puissance de calcul de l'ensemble des processeurs disponibles, RockIt a mis en place une architecture qui permet le parallélisme tout comme Tuffy [39](Noessner et al.). On peut voir dans la figure 4.8 qu'après l'instanciation, RockIt place les clauses instanciées dans une pile abstraite. Le raisonneur crée un thread par processeur et chaque clause de la pile sera traitée par un thread séparé ce qui permet d'avoir du parallélisme au niveau de la détection des clauses violées, de l'agrégation et de la transformation dans le formalisme ILP [39](Noessner et al.). On peut voir également que le parallélisme est utilisé pour la phase de résolution ILP qui utilise un algorithme « branch and bound » [39](Noessner et al.).

FIGURE 4.8 – Architecture et parallélisme de RockIt




## 4.3 Résultats

Une fois que toutes les ontologies ont été transformées en un ensemble de règles de la logique du premier ordre et puis testées avec les trois moteurs d'inférence probabilistes, nous avons obtenu un ensemble de résultats bruts. Nous avons alors entrepris une phase d'analyse concernant ces derniers.

Nous avons tout de suite repéré des tendances montrant que les temps d'Alchemy semblaient plus importants que les temps des deux autres outils au fur et à mesure que le nombre de règles augmente. Cette observation se dégage des résultats que ce soit au niveau de l'apprentissage ou au niveau de l'inférence. Nous pouvons également remarquer que pour des études de cas très petites, les temps d'exécution d'Alchemy restent raisonnables. Cependant, lorsque le nombre de règles augmentent, la croissance de la courbe est non négligeable et les temps augmentent très rapidement dépassant très vite les courbes des deux autres raisonneurs de la logique de Markov.

Les temps de Tuffy et de RockIt semblent rester sensiblement égaux bien que l'on voit qu'au fur et à mesure que le nombre de règle croit, ce dernier semble

quelque peu plus performant.

Les trois résultats ci-dessus peuvent s'expliquer en grande partie par les choix d'implémentation. En effet, alors qu'Alchemy utilise une stratégie top-down pour la phase d'instanciation et enregistre tout en mémoire centrale [25](Niu et al.,2010), Tuffy va plutôt opter pour une approche bottom-up pour exploiter totalement l'utilisation de son système de gestion de base de données relationnelle [25](Niu et al.,2010). Tuffy ne va cependant pas supporter tout le processus via la base de données mais va exécuter les procédures de recherche locales dans la mémoire centrale autant que possible [25](Niu et al.,2010). Cela va permettre de ne pas perdre beaucoup de temps dans les nombreux accès mémoires nécessaires à la recherche. Un autre aspect qui pourrait avoir une influence sur le temps d'exécution est le fait que Tuffy implémente un partitionnement du réseau de Markov produit par la phase d'instanciation. Dès lors, il est possible pour le raisonneur d'appliquer le parallélisme pour les étapes qui suivent [25](Niu et al.,2010).

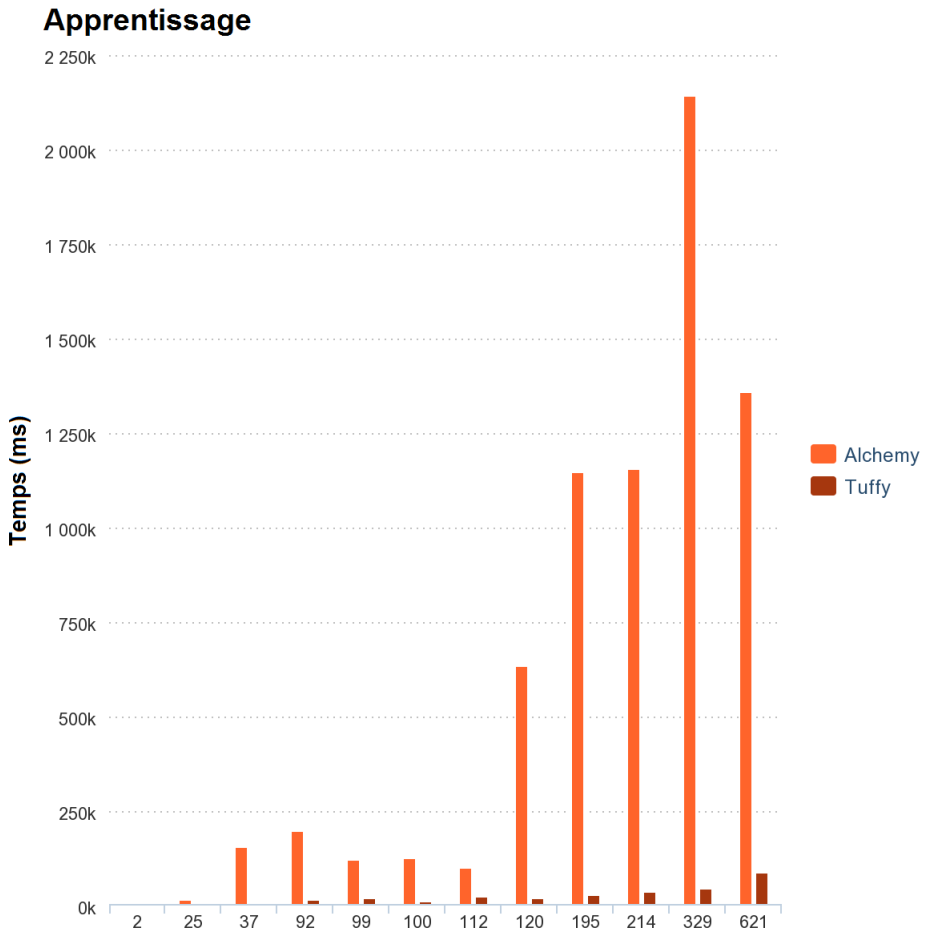
En ce qui concerne RockIt, il va assimiler le problème d'inférence à un problème d'optimisation mathématique dans lequel toutes les variables sont des entiers [39](Noessner et al.). Cet outil va ajouter une technique d'agrégation de contraintes. C'est à dire qu'au lieu de directement associer une contrainte à chaque clause instanciée, il essaie de trouver des similitudes dans les clauses pour n'obtenir qu'une seule contrainte pour les clauses similaires [39](Noessner et al.). Cela a un effet non négligeable sur les performances. RockIt va également utiliser une base de données relationnelle et appliquer le parallélisme pour l'agrégation et la résolution du problème d'optimisation [39](Noessner et al.).

Les figures 4.9 et 4.10 expriment les résultats qui viennent d'être expliqués. La première figure ne concerne qu'Alchemy et Tuffy car, comme expliqué dans la section 2.2, la version de RockIt que nous avons utilisée pour réaliser nos tests ne supportait pas encore la phase d'apprentissage de poids. La seconde figure comporte, cette fois, la comparaison des temps des trois outils. Chacun des deux graphiques ci-dessous comporte en abscisse le nombre de règles de chaque ensemble de règles de la logique du premier ordre testé. Ce facteur n'est pas optimal comme nous l'expliquerons plus tard. Cependant, pour une première comparaison en temps, il suffit amplement et permet de comparer les trois moteurs d'inférence probabilistes sur un paramètre qu'ils possèdent tous.

En outre, les résultats en temps présentés ci-dessus, seuls, ne constituent pas en soi une preuve concernant la performance des trois outils. Ils devront être analysés avec les prochains résultats afin de tirer des conclusions plus optimales.

Malgré ces résultats explicites, nous voulions trouver quel facteur influençait de manière significative l'évolution du temps. Comme nous en avons déjà discuté

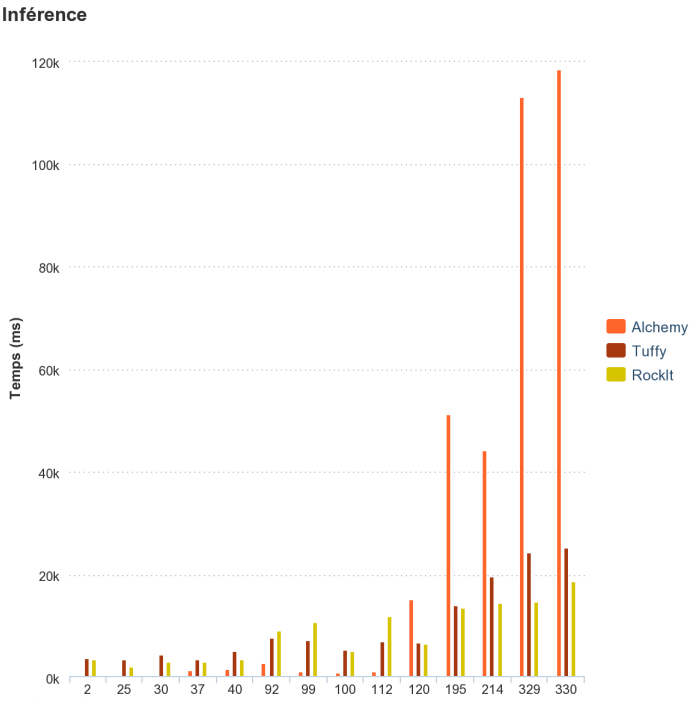
FIGURE 4.9 – Temps d'exécution des différents outils pour la phase d'apprentissage




plus haut, le nombre de règles, seul, ne permet pas d'établir une courbe de croissance parfaite pour déduire l'évolution du temps en fonction de ce paramètre. Bien que la tendance qui établit que l'un augmente en fonction du second se confirme, celle-ci ne représente pas une fonction exacte et, par conséquent, ne nous permet pas d'établir de prédictions pour des ontologies de grandes tailles.

La figure A.10 montre l'évolution du temps en fonction du nombre règles. Nous pouvons remarquer dans le cas du temps d'apprentissage d'Alchemy – choisi pour la raison de ses temps d'exécution les plus longs – que le temps en fonction du nombre de règles ne retourne pas une courbe à partir de laquelle on pourrait déterminer une fonction permettant d'extrapoler les résultats. Bien que

FIGURE 4.10 – Temps d'exécution des différents outils pour la phase d'inférence




la croissance soit assez prononcée, nous ne pouvons rien retirer de ce graphique.

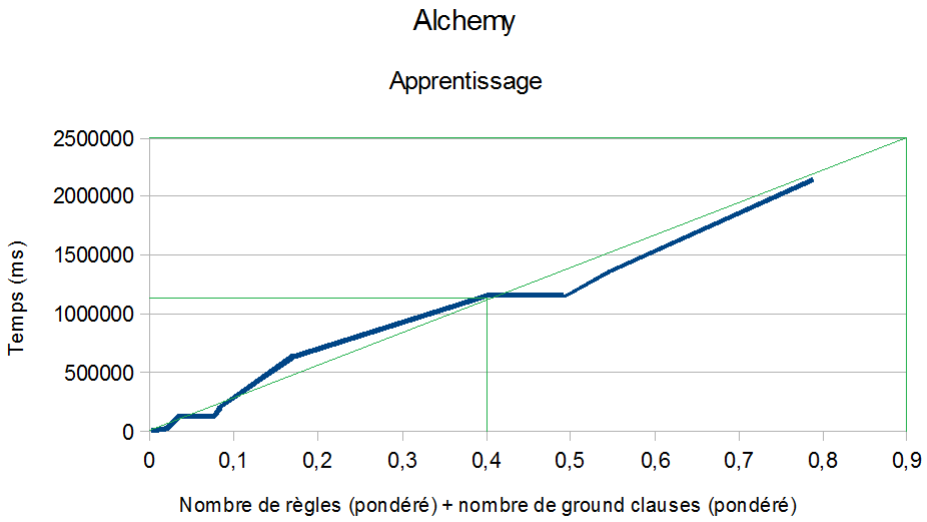
Nous avons ensuite tenté une approche plus théorique en utilisant le nombre de clauses instanciées – à savoir le nombre de clauses résultant de la phase d'instanciation – pour tenter d'établir une courbe plus représentative. Cependant, comme il est montré dans la figure A.11, les résultats retournés ne sont pas encore très significatifs. La courbe obtenue semble varier légèrement moins que la première, mais aucune extrapolation n'est encore possible à partir de l'allure que suit cette courbe.

Nous avons alors tenté de combiner plusieurs paramètres ensemble afin de vérifier s'il n'existait pas de corrélation implicites entre ceux-ci. Finalement, après une série de tests, nous avons opté pour un rapport entre ces deux derniers facteurs, à savoir le nombre de règles et le nombre de clauses instanciées. La raison de ce choix se justifie par le fait que le nombre de règles influence la phase d'instanciation produisant les clauses alors que ces dernières ont un impact sur le temps d'exécution de l'inférence. Il semble alors juste de les combiner ensemble afin d'obtenir un résultat plus proche de la réalité du processus suivi par les différents outils.

Une fois cette démarche effectuée, nous avons obtenus les graphiques ci-dessous aux figures 4.11 à 4.14.

Nous pouvons remarquer que cette fois, ces derniers ne contiennent pas de variations nettes d'un résultat à un autre. L'évolution du temps et fonction des deux facteurs précédents combinés semble suivre une croissance linéaire vers laquelle les temps assez grand semblent converger. Bien sûr, ces graphiques comportent toujours une légère marge d'erreur, mais celle-ci est plutôt dûe au facteur aléatoire des temps d'exécution et du choix des clauses instanciées lors de la phase d'inférence. Si l'on s'abstrait de cette petite marge d'erreur, nous pouvons arriver à la conclusion que l'évolution du temps en fonction d'un rapport entre le nombre de règles et le nombre de clauses instanciées est bien défini par une fonction linéaire. La découverte de ce rapport nous permet ainsi d'établir des prédictions avec un certain degré de précision quant au temps que pourrait prendre l'exécution d'un certain ensemble de règles.

FIGURE 4.11 – Prédiction de l'apprentissage pour Alchemy




La première série de tests concernant les temps d'exécution nous apportait déjà un grand nombre d'indices concernant les performances des trois raisonneurs de la logique de Markov. Cependant, après avoir mesuré les temps d'exécution de l'apprentissage et de l'inférence des différents outils et après avoir déterminé de quelle manière ce temps pouvait évoluer, il nous fallait alors comparer les résultats renvoyés par ces différents outils. Concernant ceux-ci, comme nous l'avons évoqué dans l'introduction de cette partie, nous n'avons pas voulu mesurer la pertinence des résultats car nos domaines étaient assez

FIGURE 4.12 – Prédiction de l'inférence pour Alchemy

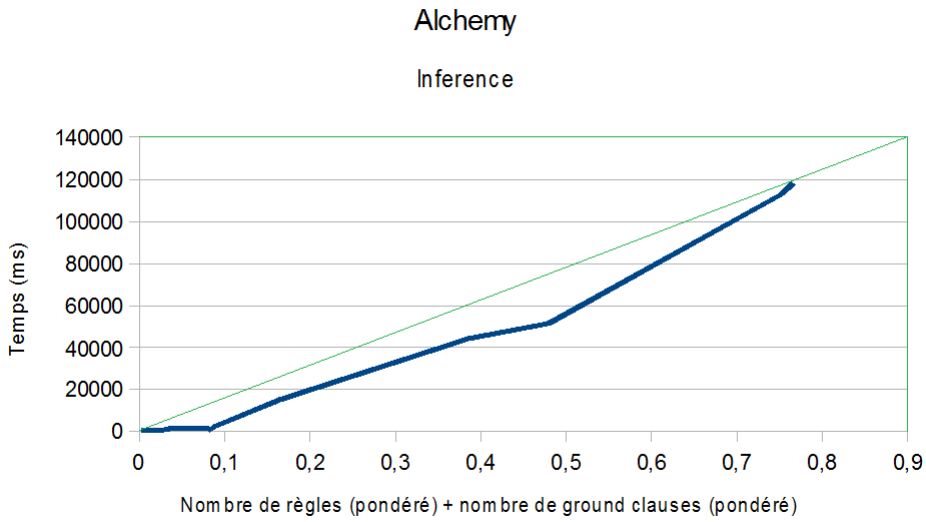
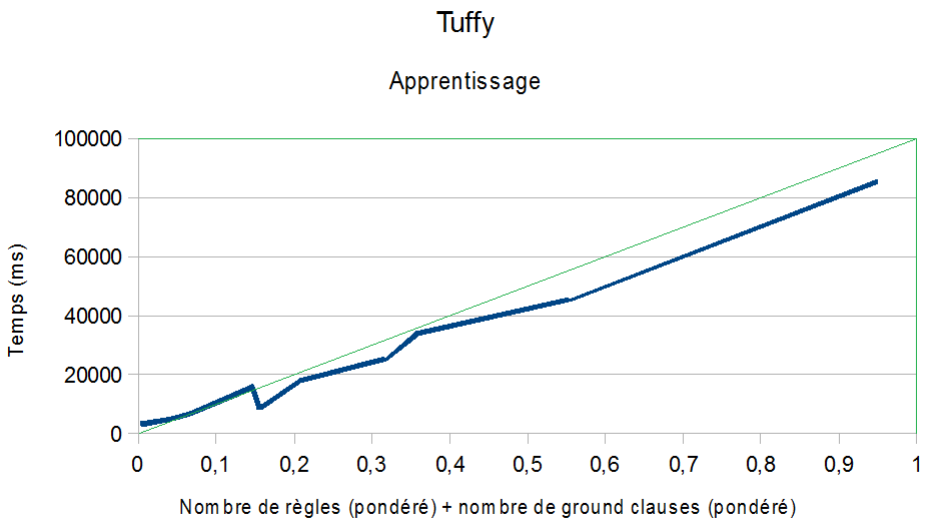
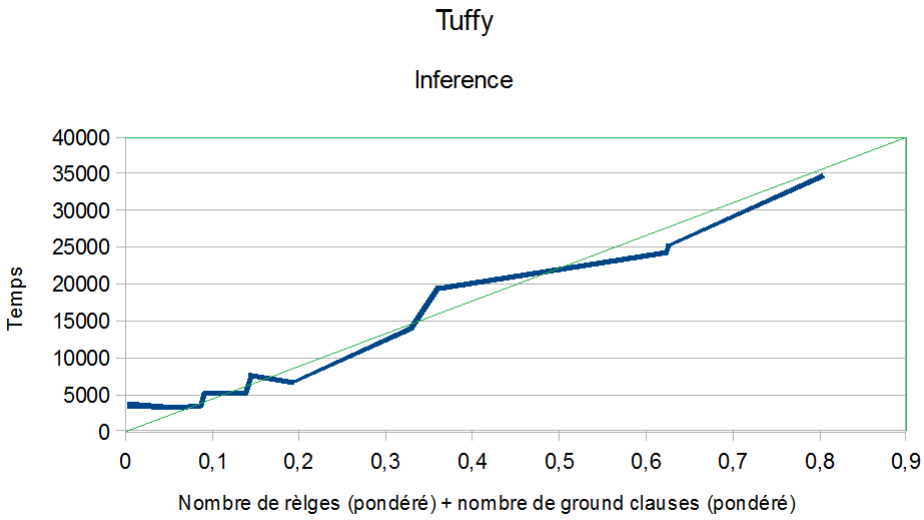



FIGURE 4.13 – Prédiction de l'apprentissage pour Tuffy




restreints et nous ne voulions pas trop utiliser le facteur « hasard » de ceux-ci. Cependant, nous avons extrait une information assez intéressante et surtout qui ne pourrait être vraiment biaisée par le facteur « hasard ». Il s'agit du nombre de résultats retournés.

FIGURE 4.14 – Prédiction de l'inférence pour



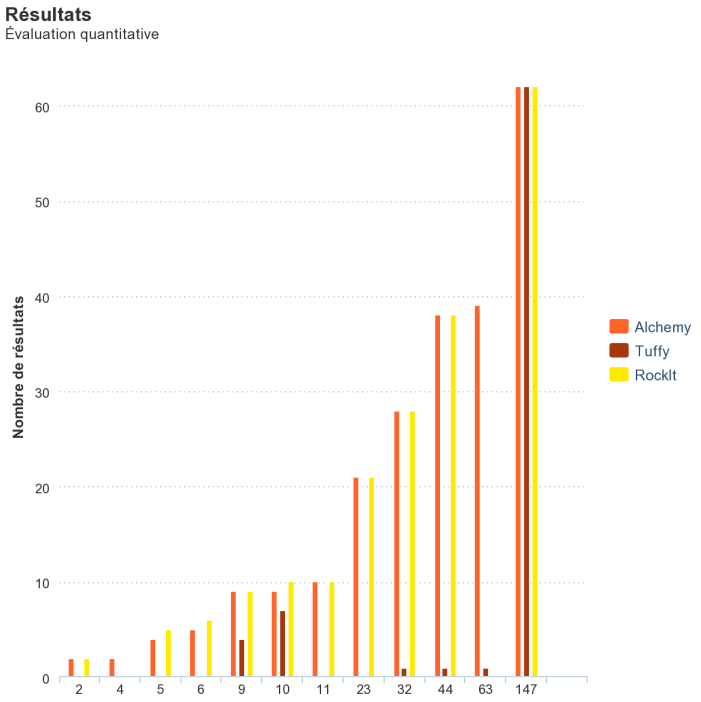

Le graphique représenté à la figure 4.15 reprend les différents nombres de résultats renvoyés par les trois raisonneurs de la logique de Markov pour la phase d'inférence en fonction du nombre d'évidences que comporte l'ontologie testée. Comme nous pouvons le constater sur ce graphique, alors que dans la plupart des cas, RockIt et Alchemy retournent un grand nombre de résultats, Tuffy n'en renvoie que très peu ou, du moins, bien moins que ses deux autres concurrents. Cette information, bien que peu significative en l'absence de contexte, nous apprend toutefois qu'Alchemy et RockIt sont plus enclin à fournir un nombre de résultats plus importants et pourraient être favorisés dans le cas d'un souhait de résultats exhaustifs pour, par exemple, obtenir également les cas limites d'un domaine d'application, reprenant même les cas très peu probables.

Pour finir l'analyse de l'efficacité des différents outils, bien que cette information ne soit pas directement liée à la phase d'apprentissage ou à la phase d'inférence, nous pouvons regarder les grammaires que proposent les trois moteurs d'inférence probabilistes étant donné que celles-ci seront nécessairement utilisées pour la production de règles.

Tout d'abord, la syntaxe proposée par Alchemy permet à l'utilisateur d'être assez souple dans ses notations. Elle est à la fois très laxiste et très compréhensive concernant les entrées de l'utilisateur.

Ensuite, celle proposée par Tuffy est déjà bien moins riche et impose un certain nombre de contraintes. C'est notamment le cas pour les clauses condition-

FIGURE 4.15 – Résultats obtenus par les différentes outils




nelles pour lesquelles l'antécédent n'accepte que des conjonctions alors que le conséquent quant à lui ne tolère que des disjonctions. L'utilisateur doit ainsi être plus prudent lorsqu'il utilise la notation de Tuffy afin de ne pas commettre d'erreur de syntaxe.

Enfin, la syntaxe proposée par RockIt est largement restreinte et oblige l'utilisateur à connaître des notations très spécifiques à ce langage. C'est par exemple le cas de la clause « Il existe » qui nécessite une notation non triviale et séparée en deux parties au sein de la même règle. L'utilisateur doit à ce moment être très attentif et doit respecter une notation très spécifique à cet outil.

Bien que ce détail de syntaxe n'ait pas beaucoup d'importance concernant les résultats et leurs temps, il peut s'agir d'un facteur important dans la phase d'implémentation car Alchemy est beaucoup moins contraignant que ses concurrents. Cette information devrait alors être prise en compte dans le choix de l'outil surtout si les développeurs ne sont habitués à aucun des trois outils.

En conclusion, nous pourrions dire qu'il n'y a pas un seul outil à privilégier dans tous les cas, mais que chaque outil a ses points forts et points faibles suivant

le contexte dans lequel on l'utilise.

Alchemy est très utilisé sur le marché et permet donc d'améliorer l'interopérabilité avec d'autres programmes. Sa syntaxe est également très étendue et offre ainsi des possibilités importantes pour le développeur, notamment quant à sa prise en main de l'outil. Il offre également des facilités non négligeables pour l'utilisateur qui doit s'adapter à sa syntaxe. Il est également à noter que les résultats retournés sont relativement riches quantitativement. Cependant, les temps retournés par cet outil croissent très vite et il devient assez rapidement exclu d'utiliser celui-ci pour des domaines très peuplés, surtout si l'exécution doit se dérouler en temps réel.

Tuffy possède une syntaxe plus contraignante et est beaucoup moins utilisé sur le marché. La richesse quantitative de ces résultats est également assez faible. En effet, Tuffy ne fournissant pas souvent un nombre important de résultats, il pourrait être à exclure pour des domaines ayant absolument besoin de connaitre l'ensemble des possibilités d'une requête, incluant les cas limites. Cependant, la force de ses choix d'implémentation lui offre une efficacité drastiquement améliorée en termes de temps d'exécution par rapport à Alchemy lorsqu'il s'agit de domaines de grande taille.

RockIt, quant à lui, est naissant et n'est donc pas encore vraiment utilisé à l'heure actuelle. Le processus d'apprentissage de poids n'est pas encore implémenté et sa syntaxe contraint l'utilisateur à une notation très spécifique, ce qui peut être un argument négatif en cas d'utilisation ou de développement, surtout pour des développeurs novices en la matière. Cependant, cet outil renvoie un grand nombre de résultats et cela dans des temps très peu élevés. Concernant ces deux derniers points, RockIt devrait être un choix intéressant pour des domaines travaillant en temps réel et nécessitant un nombre non négligeable de résultats.

Nous pourrions ainsi conclure en disant que pour des domaines de petites tailles, Alchemy semblerait mieux convenir de par sa rapidité raisonnable, sa richesse et son utilisabilité. Cependant, dès que l'ensemble de règles augmente Tuffy ou RockIt devrait être à privilégier. RockIt semble un peu plus rapide que son homologue et retourne également beaucoup de résultats. Il est cependant plus contraignant et est encore dans une phase naissante, ce qui engendre qu'il comporte encore de nombreux bugs et sera encore amené à évoluer au cours du temps. Il faudrait alors plutôt considérer Tuffy pour des projets plus importants car la maturité de celui-ci a déjà atteint un haut niveau et a déjà fait ses preuves. Il est toutefois intéressant de suivre de près les améliorations de RockIt car il est en pleine phase de développement et semble contenir des idées prometteuses.

Un tableau récapitulatif des conclusions que nous avons réalisées sur les trois moteurs d'inférence probabilistes se trouvent à la figure 4.16.

FIGURE 4.16 – Tableau comparatif des conclusions concernant les outils

| | Alchemy | Tuffy | RockIt |
|---|---|---|---|
| Rapidité | - | ++ | ++ |
| Nombre de résultats | + | - | + |
| Facilité de la grammaire | ++ | + | - |

# Conclusion

Pour répondre aux objectifs fixés dans ce mémoire, nous avons créé un outil, ArThUR [1] (Ortiz et al., 2014), pour lequel un article scientifique sera présenté à la conférence d'Amantea en Italie du 27 au 31 octorbre 2014 [1]. Cet outil est destiné à travailler avec des ontologies et à inférer sur la base de connaissances de cette dernière. Les calculs probabilistes réalisés peuvent alors faire l'objet de statistiques afin de déterminer lequel des trois raisonneurs de la logique de Markov que nous avons choisi – à savoir Alchemy, Tuffy et RockIt – est le meilleur dans tel ou tel cas.

Pour commencer, notre programme permet à un utilisateur du projet ORTHOGEN de transformer la connaissance d'une ontologie en un réseau logique de Markov qui sera représenté grâce à un ensemble de règles de la logique du premier ordre auxquelles seront attachés des poids.

Cette étape, comme nous l'avons expliqué tout au long de ce travail, offre à l'utilisateur la possibilité de manipuler la base de connaissances de l'ontologie sous une forme qui pourra être comprise par les moteurs d'inférence probabilistes. Ainsi, il est alors possible d'inférer sur les éléments des ontologies grâce à la transformation automatique de cette dernière par notre programme. De plus, l'utilisateur d'ORTHOGEN dispose d'un certain nombre de fonctionnalités lui permettant d'améliorer efficacement et facilement l'ensemble de règles qu'il manipule. Il peut ainsi, à sa guise, altérer ces règles afin de faire varier les résultats qui en découleront lorsque seront appelés les trois raisonneurs de la logique de Markov.

En plus de la facilité d'utilisation et de modification que l'utilisateur possède, l'outil que nous proposons permet également aux professionnels en informatique d'établir un dialogue amélioré avec les médecins de Mont-Godinne grâce à un système de traduction. La simplicité de transformation des règles exprimées dans la logique du premier ordre en phrases exprimées en anglais a pour objectif d'octroyer un confort lors des réunions avec les spécialistes médicaux car tous les membres présents à celles-ci pourront désormais communiquer à partir de faits qu'ils seront tous capables de comprendre.

1. http ://www.onthemove-conferences.org/index.php/swws2014

Outre ces aides fournies à l'utilisateur, nous avons, dans une démarche évolutive, proposé un API comportant les fonctionnalités les plus importantes et pertinentes de notre outil afin de permettre à d'autres programmes ou processus d'intégrer nos fonctions au sein des leurs. Il se pourrait même que le projet final d'ORTHOGEN décide d'utiliser ces dernières s'il ne désire pas l'intégralité du programme que nous fournissons dans ce mémoire. D'autre part, cette API peut également être considérer comme un plus s'il s'avérait que notre projet soit repris et amélioré dans le futur. Une application web pourrait ainsi économiser un certain temps s'il elle utilisait notre API à la place de tout devoir réimplémenter.

Grâce au réseau logique de Markov formé à partir des règles récupérées d'une ontologie, nous avons également pu mettre en place des mécanismes d'apprentissage et d'inférence de ce réseau. Ces mécanismes nous permettent, grâce aux trois moteurs d'inférence probabilistes, de nous rapprocher le plus possible de la vérité provenant de la connaissance d'une ontologie lorsque nous inférons. L'approche de cette vérité est possible car nous tirons profit des probabilités octroyées par les déductions des réseaux logiques de Markov.

Ainsi, après avoir obtenus ces probabilités avec les trois raisonneurs de la logique de Markov que nous avons intégrés dans notre projet, nous pouvons nous appuyer sur des statistiques pour réaliser notre second grand objectif, à savoir la sélection du moteur d'inférence probabiliste le plus adéquat dans un contexte donné. Le module de statistiques de notre outil nous permet d'obtenir les temps d'exécution, ainsi que les résultats obtenus par respectivement Alchemy, Tuffy et RockIt. Les temps d'exécution sont importants afin de savoir si les raisonneurs ne prennent pas trop longtemps lors de leurs phases d'apprentissage ou d'inférence, alors que les résultats retournés nous indiquent si les raisonneurs sont plutôt exhaustifs ou limités dans les résultats qu'ils renvoient.

Après avoir réalisé notre phase d'analyse grâce à une quinzaine de cas de tests recouvrant une grande partie des possibilités entre le nombre de règles et le nombre d'évidences, nous avons obtenus plusieurs graphiques nous indiquant, d'une part, comment se comportait les outils suite à un ensemble de règles reçu en entrée et, d'autre part, nous pouvions également établir des prédictions quant au temps que pourrait prendre une phase d'apprentissage ou d'inférence avec un outil donné.

Il ne nous restait plus qu'à tirer des conclusions quant à l'outil à choisir dans une situation bien définie. En plus des temps d'exécution et des résultats obtenus, nous avons également choisi de considérer la facilité avec laquelle le développement des règles pouvait être réalisé. Pour cela, nous avons pris en compte un élément supplémentaire. Il s'agit de la facilité d'apprentissage de la grammaire de chaque raisonneur, ainsi que de son utilisabilité.

Ainsi, avec l'ajout de ce dernier point à prendre en compte, nous pouvons définir quel outil est le meilleur en phase de production, mais également en phase de développement. Ainsi, les résultats que nous avons obtenus sont les suivants.

Tout d'abord, Alchemy est le meilleur choix en phase de développement. En effet, le laxisme, mais aussi la trivialité de la grammaire qu'il utilise permet à l'utilisateur d'apprendre cette dernière rapidement et d'éviter les erreurs de syntaxe qui pourraient apparaître dans des grammaires trop strictes. A l'inverse, les deux autres outils ne sont pas aussi bons dans ce domaine. Tuffy arrive toutefois en seconde position si nous devons définir un classement. Sa grammaire est beaucoup plus stricte et il devient plus difficile de ne pas réaliser d'erreurs lors des exécutions. Cependant, sa grammaire reste assez triviale dans l'ensemble et ne comporte pas de termes trop particuliers. Enfin, RockIt arrive en dernière position en ce qui concerne la facilité de prise en main pour un développeur. En effet, ce dernier raisonneur propose une syntaxe à la fois fort stricte et très spécifique. Ainsi, certaines structures de règles longues et complexes deviennent difficiles à réaliser sans l'aide d'un automate. Celui-ci pose ainsi le plus de problèmes lors de la phase de développement.

Concernant la phase de production, les résultats sont cependant tout autres. Tout d'abord, Alchemy devient vite contraignant lorsque les domaines de règles augmentent de manière trop importante. Alors que pour quelques règles et évidences, ce dernier obtient des temps tout à fait raisonnables. La situation change drastiquement dès que le nombre de règles dépasse la centaine. A ce moment, les temps d'exécution augmentent énormément et il devient de plus en plus contraignant de devoir attendre les résultats de cet outil. Au contraire, ses deux homologues gardent des résultats relativement raisonnables au fur et à mesure que le nombre de règles et d'évidences croissent. Lorsque ce nombre est grand, nous pouvons remarquer que RockIt semble un rien plus performant que Tuffy. Cependant, cette différence n'est pas aussi significative que celle qui les distance d'Alchemy. Ainsi, si l'on souhaite travailler sur des petits ensembles ou si les processus qui sont amenés à tourner peuvent s'effectuer endéans un long moment, Alchemy ne pose pas de problème. Cependant, dès que l'on doit inférer sur des plus grosses bases de connaissances et que les résultats doivent être obtenus en temps réel ou simplement dans un délai raisonnablement court, il faut à ce moment-là se tourner vers Tuffy ou RockIt.

Cependant, les temps d'exécution ne signifient pas tout. Outre ce point, le nombre de résultats obtenus peut également être important. Ainsi, alors qu'Alchemy et RockIt semblent obtenir un grand nombre de résultats lors de nos expériences, Tuffy en calcule généralement moins d'une dizaine. Cette information peut être intéressante, en particulier dans le domaine médical, car elle signifie que Tuffy ne se contente que des cas principaux alors que les deux autres raisonneurs semblent envisager un plus grand nombre de cas possibles. Ainsi,

dans le cadre du projet d'ORTHOGEN, il pourrait être intéressant d'obtenir certains cas limites. Tuffy pourrait à ce moment-là être à exclure.

En conclusion, il n'y a pas un outil meilleur que les autres. Chacun d'entre eux possède des avantages et des inconvénients. Cependant, nous pourrions nous tracasser plus précisément du cas de l'ontologie d'ORTHOGEN, que nous avons, par ailleurs, utilisé dans nos tests. Celle-ci possède une base de connaissances assez conséquente et, dans un monde idéal, les médecins ne devraient pas attendre un trop long moment avant d'obtenir les résultats qu'ils souhaitent. Ainsi, cette contrainte de temps pourrait exclure d'entrée Alchemy. On pourrait alors se diriger vers deux solutions potentielles. D'une part, Tuffy et, d'autre part, RockIt. Ce second raisonneur en plus d'offrir un temps d'exécution aussi bon, voire meilleur que celui de Tuffy semble proposer plus de résultats. Cependant, celui-ci est encore en phase de développement et n'est pas encore très stable. Il pourrait alors poser problème pour un domaine aussi minutieux que celui de la médecine. Le choix semblerait alors se diriger vers Tuffy.

Dès lors, nous pourrions envisager un scénario dans lequel les règles seraient créées à partir de la syntaxe d'Alchemy et où l'exécution, après transformation de la syntaxe, serait réalisée avec l'outil Tuffy.

Un autre point qu'il ne faut pourtant pas oublier est que l'hôpital de Mont-Godinne utilise actuellement Alchemy. Bien que l'utilisation de cet outil se fasse dans un cadre expérimentale, changer ses habitudes pourrait à nouveau être contraignant. Dès lors, ORTHOGEN pourrait à nouveau s'interfacer avec la syntaxe d'Alchemy pour ses échanges avec l'hôpital avant d'inférer avec Tuffy après un changement de syntaxe.

Ces deux scénarios semblent possibles à l'heure actuelle. Cependant, il ne faut certainement pas laisser de côté RockIt qui devrait être suivi régulièrement pour, plus tard, lorsqu'il serait développé dans une version stable, être intégré dans les processus du projet de par sa puissance qui semble la plus prometteuse en termes d'efficacité. Le suivi de cet outil est alors primordial pour la suite du projet.

Nous venons de vous montrer les avantages et inconvénients des trois outils que nous avons analysés. Nous avons proposé également plusieurs scénarios pour la suite du projet ORTHOGEN. Cependant, comme nous l'avons plusieurs fois évoqué au cours de ce mémoire, aucun scénario n'est définitivement établi, mais c'est le contexte et l'environnement dans lequel se situera le projet qui décidera, à ce moment-là, quel choix semble être le plus judicieux.

# Perspectives

Ce travail, bien qu'actuellement dans un état final, pourrait recevoir un certain nombre d'améliorations dans le futur. Celles-ci rassembleraient plusieurs objectifs. Nous pourrions tout d'abord améliorer le module de statistiques grâce à une optimisation des résultats obtenus ou encore grâce à une amélioration des temps d'exécution. Nous pourrions ensuite améliorer l'utilisabilité de notre outil en proposant diverses modifications visant à améliorer l'expérience de l'utilisateur. Enfin, nous pourrions proposer d'étendre la portée d'utilisation de notre outil. Nous vous listons ci-dessous les différents scénarios auxquels nous avons pensé.

Tout d'abord, comme il a été évoqué dans la section 2.3 traitant des limites de ce mémoire, nous pourrions tenter d'interfacer notre programme directement avec un organisme contenant des données réelles. Il pourrait, par exemple, s'agir de l'hôpital de Mont-Godinne qui nous fournirait des données anonymisées de ses patients. Cette démarche nous permettrait d'augmenter de manière considérable le nombre de règles, mais surtout le nombre d'évidences que nous traiterions lors de nos scénarios de tests. Ainsi, le fait de manipuler des domaines de connaissance nettement plus grand nous permettrait de tester les limites en temps des trois outils lorsque les ensembles de règles et d'évidences croissent de manière importante.

Ensuite, dans la lignée de cette première idée, nous pourrions réaliser une autre des limites que nous avions déjà décrites dans la section 2.3. Il s'agirait, cette fois, de nous connecter directement à la base de données de Tuffy ou de RockIt afin de pouvoir mémoriser la phase d'instanciation et d'ainsi gagner un temps important lors de la phase d'exécution. Cette amélioration complèterait la première évoquée plus haut, car, si nous testons des échantillons plus grands, les temps augmenteront de manière conséquente. Il serait alors intéressant de diminuer ces derniers grâce à des stratégies permettant de nous connecter directement aux outils concernés.

La dernière des limites que nous avions évoquées concerne l'ajout de l'analyse de la qualité des résultats. Cette tâche est, comme nous l'avons déjà évoqué, très compliquée car elle nécessite que nous connaissions déjà les résultats à obtenir

avant de lancer notre phase de tests. Cependant, si nous arrivions à mettre la main sur de tels résultats, notre analyse des différents moteurs d'inférence probabilistes serait grandement améliorée. Nous pourrions calculer l'écart entre, d'une part, les résultats retournés par les trois raisonneurs et, d'autre part, les proportions calculées sur un ensemble de cas réels. Cet ensemble de cas pourrait, par exemple, provenir d'une base de données contenant une série de résultats médicaux concernant les différents patients d'un l'hôpital.

Nous allons maintenant évoquer d'autres pistes que celles qui ont été décrites dans la section traitant des limites des objectifs de ce mémoire. Tout d'abord, une optimisation qui semble assez anodine mais pourrait être d'une grande utilité pour des ensembles de règles importants est l'ajout de moteur de recherche de prédicats. A l'heure actuelle, lorsqu'un utilisateur veut rechercher un ou plusieurs prédicats particuliers, il est obligé de lire règle par règle l'ensemble de celles-ci. L'ajout d'un tel moteur pourrait alors lui faire économiser un temps précieux lors de ses recherches.

Une autre idée intéressante mais, cette fois, plus difficile à mettre en place serait la création d'une interface de notre programme destinée au personnel médical – ou dans un but d'expansion de notre outil, à une personne ne maitrisant pas la logique ou les différentes technologies que nous utilisons. En effet, à l'heure actuelle, notre outil offre un module de traduction permettant l'analyse par un expert non informaticien. Cependant, celui-ci doit obligatoirement être supervisé par un expert en informatique, car le reste de l'outil n'est pas très ergonomique en ce qui le concerne. Une idée à creuser pourrait alors être la création d'une interface qui serait en tout point compréhensible pour une personne ne maîtrisant pas l'informatique ou la logique. Nous devrions bien sûr retravailler notre manière de présenter l'information à l'utilisateur, mais nous pourrions ainsi toucher un public beaucoup plus large.

Enfin, en nous aidant de notre API, nous pourrions proposer les fonctionnalités que nous avons expliquées tout au long de ce mémoire sur un site web. Nous voyons cette démarche comme un service rendu gratuitement à la société. En effet, nous fournirions nos possibilités de calcul gratuitement sur Internet. Cela pourrait, d'une part, nous offrir des échantillons de tests plus conséquents si les utilisateurs acceptent qu'on nous analysions leurs données. De plus, nous pourrions obtenir des retours plus précis quant à l'efficacité et l'utilisabilité de notre outil.

Nous venons de vous partager nos différentes perspectives pour l'avenir de ce projet. Cet outil bien que dans une version tout à fait utilisable à l'heure actuelle pourra ainsi se voir optimiser dans les prochaines années.

# Table des figures

# Annexe A

# Informations supplémentaires

## A.1 Annexe de l'état de l'art

### A.1.1 Algèbre de Boole

Les connecteurs de la logique du premier ordre répondent des mêmes priorités que les opérateurs de l'algèbre de Boole :

- Associativité :
  - $(A \vee B) \vee C = A \vee (B \vee C) = A \vee B \vee C$
  - $(A \wedge B) \wedge C = A \wedge (B \wedge C) = A \wedge B \wedge C$
- Commutativité :
  - $A \vee B = B \vee A$
  - $A \wedge B = B \wedge A$
- Distributivité :
  - $A \wedge (B \vee C) = A \wedge B \vee A \wedge C$
  - $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$
- Idempotence :
  - $A \vee A \vee [...] \vee A = A$
  - $A \wedge A \wedge [...] \wedge A = A$
- Éléments neutres :
  - $A \vee Faux = A$
  - $A \wedge Vrai = A$
- Absorption :
  - $Faux \wedge A = Faux$
  - $Vrai \vee A = Vrai$

Pour pouvoir interpréter correctement une formule de la logique du premier ordre il faut savoir que l'ensemble des connecteurs logiques se lisent de gauche à droite (associativité à gauche) sauf l'implication qui se lit de droite à gauche (as-

sociativité à droite). Il faut également connaître l'ordre de priorité des différents connecteurs. Si on considère l'opérateur de précédence $\prec$ on a que :

$$\Leftrightarrow \prec \Rightarrow \prec \vee \prec \wedge \prec \neg$$

On a, par exemple, que :
- $A \wedge B \wedge C$ se lit $(A \wedge B) \wedge C$
- $A \Rightarrow B \Rightarrow C$ se lit $A \Rightarrow (B \Rightarrow C)$
- $\neg A \Leftrightarrow B \wedge C$ se lit $(\neg A) \Leftrightarrow (B \wedge C)$

### A.1.2 Réseau logique de Markov

$D = \{Peter, Simon, Mark\}$

$$\begin{array}{lcl}
F_1 : & \neg isTutor(x,y) \vee BacStudent(y) & w_1 \\
F_2 : & \neg givesCourses(x,y) \vee PersonalMember(x) & w_2 \\
F_3 : & \neg Researcher(x) \vee PersonalMember(x) & w_3 \\
F_4 : & \neg MasterStudent(x) \vee Student(x) & w_4 \\
F_5 : & \neg givesCourses(x,y) \vee Student(y) & w_5 \\
F_6 : & \neg BacStudent(x) \vee Student(x) & w_6 \\
F_7 : & \neg isTutor(x,y) \vee hasSocialInterest(x) & w_7 \\
F_8 : & \neg hasSocialInterest(x) \vee Professor(x) & w_8 \\
F_9 : & \neg Professor(x) \vee PersonalMember(x) & w_9
\end{array}$$

$$\begin{array}{lc}
E_1 : & BacStudent(Mark) \\
E_2 : & PersonalMember(Peter) \\
E_3 : & hasSocialInterest(Simon)
\end{array}$$

**Instanciation**

$D^2 = D \times D = \{\{Peter, Peter\}, \{Peter, Simon\}, \{Peter, Mark\}, \{Simon, Simon\}, \{Simon, Peter\}, \{Simon, Mark\}, \{Mark, Mark\}, \{Mark, Peter\}, \{Mark, Simon\}\}$

$$\begin{array}{lcl}
F_{1,1} : & \neg isTutor(Peter, Peter) \vee BacStudent(Peter) & w_1 \\
F_{1,2} : & \neg isTutor(Peter, Simon) \vee BacStudent(Simon) & w_1 \\
F_{1,3} : & \neg isTutor(Peter, Mark) \vee BacStudent(Mark) & w_1 \\
F_{1,4} : & \neg isTutor(Simon, Peter) \vee BacStudent(Peter) & w_1 \\
F_{1,5} : & \neg isTutor(Simon, Simon) \vee BacStudent(Simon) & w_1 \\
F_{1,6} : & \neg isTutor(Simon, Mark) \vee BacStudent(Mark) & w_1 \\
F_{1,7} : & \neg isTutor(Mark, Peter) \vee BacStudent(Peter) & w_1 \\
F_{1,8} : & \neg isTutor(Mark, Simon) \vee BacStudent(Simon) & w_1 \\
F_{1,9} : & \neg isTutor(Mark, Mark) \vee BacStudent(Mark) & w_1
\end{array}$$

$$
\begin{array}{lrl}
F_{2,1}: & \neg givesCourses(Peter, Peter) \vee PersonalMember(Peter) & w_2 \\
F_{2,2}: & \neg givesCourses(Peter, Simon) \vee PersonalMember(Peter) & w_2 \\
F_{2,3}: & \neg givesCourses(Peter, Mark) \vee PersonalMember(Peter) & w_2 \\
F_{2,4}: & \neg givesCourses(Simon, Peter) \vee PersonalMember(Simon) & w_2 \\
F_{2,5}: & \neg givesCourses(Simon, Simon) \vee PersonalMember(Simon) & w_2 \\
F_{2,6}: & \neg givesCourses(Simon, Mark) \vee PersonalMember(Simon) & w_2 \\
F_{2,7}: & \neg givesCourses(Mark, Peter) \vee PersonalMember(Mark) & w_2 \\
F_{2,8}: & \neg givesCourses(Mark, Simon) \vee PersonalMember(Mark) & w_2 \\
F_{2,9}: & \neg givesCourses(Mark, Mark) \vee PersonalMember(Mark) & w_2
\end{array}
$$

$$
\begin{array}{lrl}
F_{3,1}: & \neg Researcher(Peter) \vee PersonalMember(Peter) & w_3 \\
F_{3,2}: & \neg Researcher(Simon) \vee PersonalMember(Simon) & w_3 \\
F_{3,3}: & \neg Researcher(Mark) \vee PersonalMember(Mark) & w_3
\end{array}
$$

$$
\begin{array}{lrl}
F_{4,1}: & \neg MasterStudent(Peter) \vee Student(Peter) & w_4 \\
F_{4,2}: & \neg MasterStudent(Simon) \vee Student(Simon) & w_4 \\
F_{4,3}: & \neg MasterStudent(Mark) \vee Student(Mark) & w_4
\end{array}
$$

$$
\begin{array}{lrl}
F_{5,1}: & \neg givesCourses(Peter, Peter) \vee Student(Peter) & w_5 \\
F_{5,2}: & \neg givesCourses(Peter, Simon) \vee Student(Simon) & w_5 \\
F_{5,3}: & \neg givesCourses(Peter, Mark) \vee Student(Mark) & w_5 \\
F_{5,4}: & \neg givesCourses(Simon, Peter) \vee Student(Peter) & w_5 \\
F_{5,5}: & \neg givesCourses(Simon, Simon) \vee Student(Simon) & w_5 \\
F_{5,6}: & \neg givesCourses(Simon, Mark) \vee Student(Mark) & w_5 \\
F_{5,7}: & \neg givesCourses(Mark, Peter) \vee Student(Peter) & w_5 \\
F_{5,8}: & \neg givesCourses(Mark, Simon) \vee Student(Simon) & w_5 \\
F_{5,9}: & \neg givesCourses(Mark, Mark) \vee Student(Mark) & w_5
\end{array}
$$

$$
\begin{array}{lrl}
F_{6,1}: & \neg BacStudent(Peter) \vee Student(Peter) & w_6 \\
F_{6,2}: & \neg BacStudent(Mark) \vee Student(Mark) & w_6 \\
F_{6,3}: & \neg BacStudent(Simon) \vee Student(Simon) & w_6
\end{array}
$$

$$
\begin{array}{lrl}
F_{7,1}: & \neg isTutor(Peter, Peter) \vee hasSocialInterest(Peter) & w_7 \\
F_{7,2}: & \neg isTutor(Peter, Simon) \vee hasSocialInterest(Peter) & w_7 \\
F_{7,3}: & \neg isTutor(Peter, Mark) \vee hasSocialInterest(Peter) & w_7 \\
F_{7,4}: & \neg isTutor(Simon, Peter) \vee hasSocialInterest(Simon) & w_7 \\
F_{7,5}: & \neg isTutor(Simon, Simon) \vee hasSocialInterest(Simon) & w_7 \\
F_{7,6}: & \neg isTutor(Simon, Mark) \vee hasSocialInterest(Simon) & w_7 \\
F_{7,7}: & \neg isTutor(Mark, Peter) \vee hasSocialInterest(Mark) & w_7 \\
F_{7,8}: & \neg isTutor(Mark, Simon) \vee hasSocialInterest(Mark) & w_7 \\
F_{7,9}: & \neg isTutor(Mark, Mark) \vee hasSocialInterest(Mark) & w_7
\end{array}
$$

$$
\begin{array}{lrl}
F_{8,1}: & \neg hasSocialInterest(Peter) \vee Professor(Peter) & w_8 \\
F_{8,2}: & \neg hasSocialInterest(Mark) \vee Professor(Mark) & w_8 \\
F_{8,3}: & \neg hasSocialInterest(Simon) \vee Professor(Simon) & w_8
\end{array}
$$

$$
\begin{array}{lll}
F_{9,1}: & \neg Professor(Peter) \vee PersonalMember(Peter) & w_9 \\
F_{9,2}: & \neg Professor(Mark) \vee PersonalMember(Mark) & w_9 \\
F_{9,3}: & \neg Professor(Simon) \vee PersonalMember(Simon) & w_9
\end{array}
$$

Nombre total de clauses instanciées :

$$N_c = 3^2 + 3^2 + 3^1 + 3^1 + 3^2 + 3^1 + 3^2 + 3^1 + 3^1 = 51$$

Nombre total d'atomes instanciés :

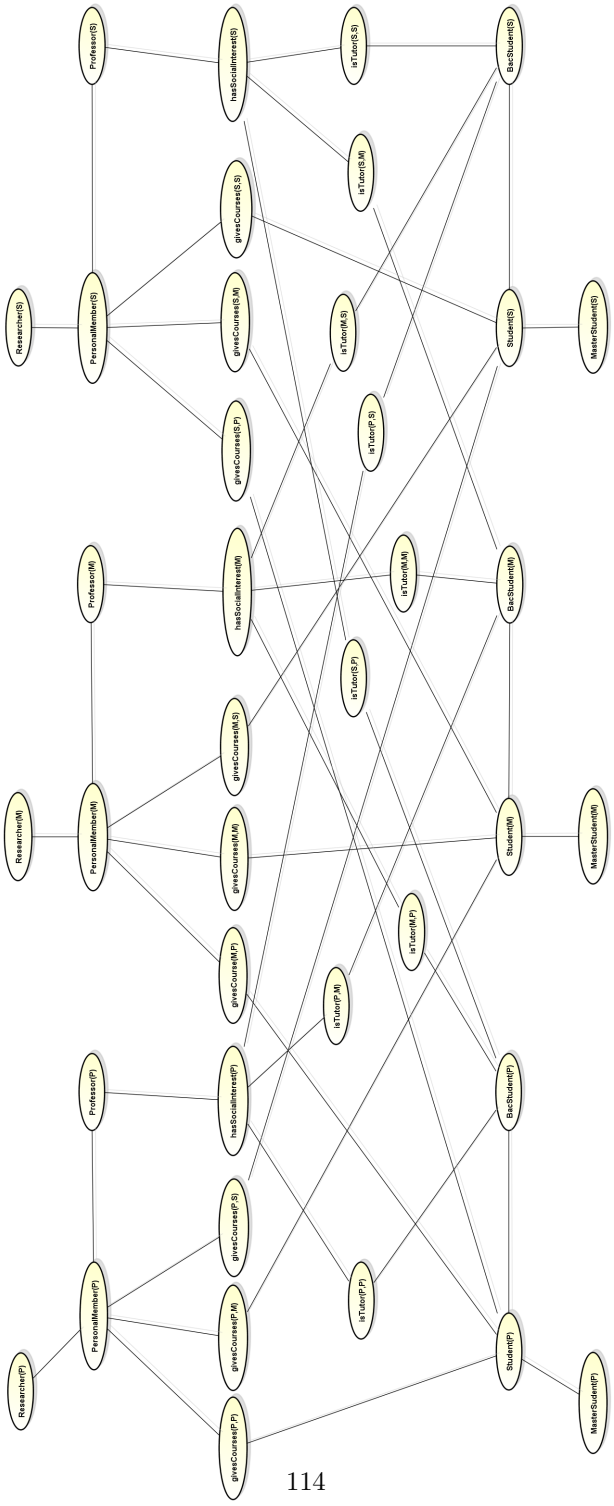
$N_a = \sum_i |D|^{n_i}$ où $n_i$ est le nombre de termes dans le prédicat $P_i$

$$N_a = 3^2 + 3^1 + 3^2 + 3^1 + 3^1 + 3^1 + 3^1 + 3^1 + 3^1 = 39$$

Le nombre de mondes possibles différents peut etre calculé à partir des atomes instanciés :

Nombre de mondes possibles $= 2^{N_a} = 2^{39} = 549755813888$ mondes possibles

FIGURE A.1 – Exemple de réseau de Markov instancié



## A.2 Annexe de la partie développement

FIGURE A.2 – Choix de l'ontologie

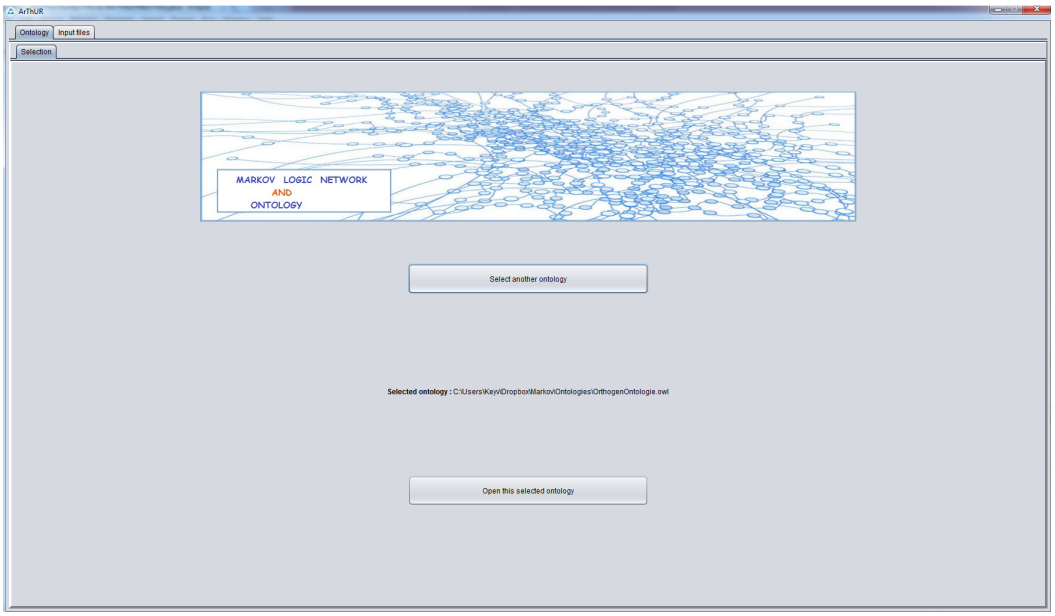



FIGURE A.3 – Paramètre lors de la transformation d'une ontologie

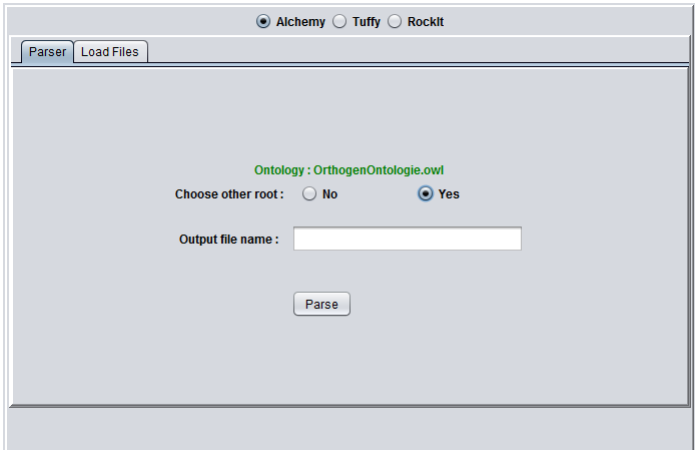


FIGURE A.4 – Choix de la ou des racines

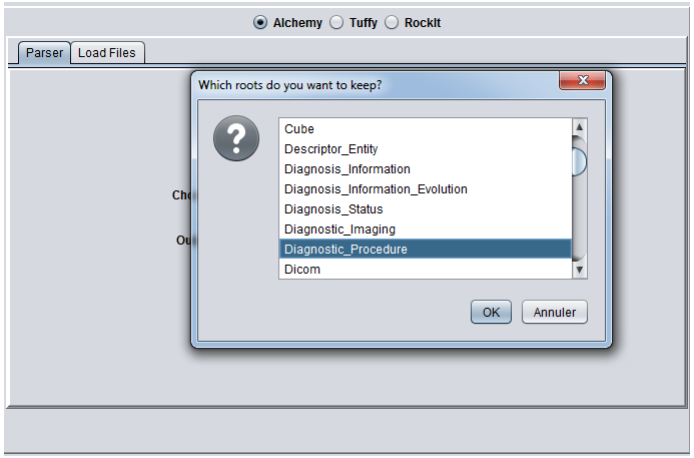



FIGURE A.5 – Choix de l'inclusion ou de l'exclusion des racines sélectionnées

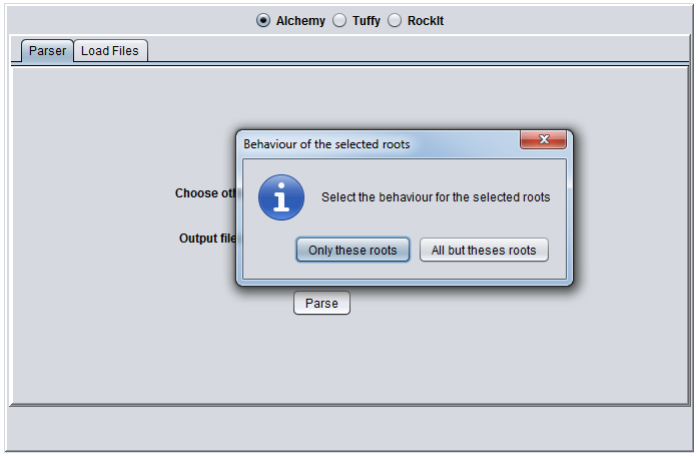


FIGURE A.6 – Suppression de déclarations toujours utilisées

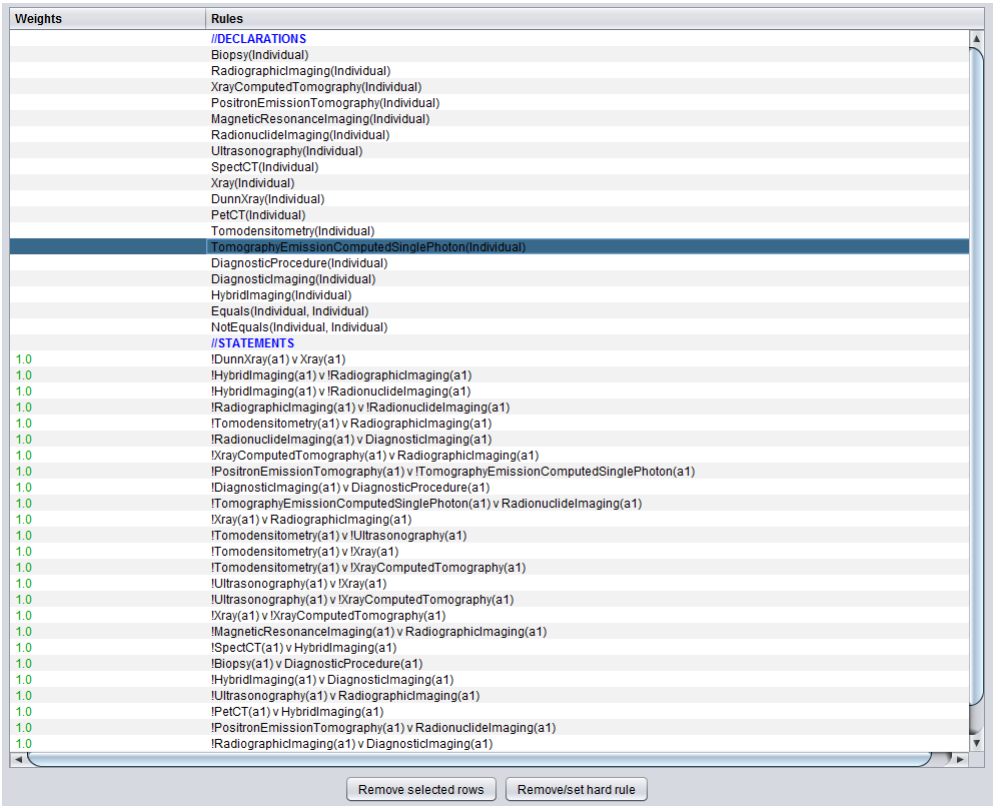


FIGURE A.7 – Suppression de déclarations toujours utilisées (2)

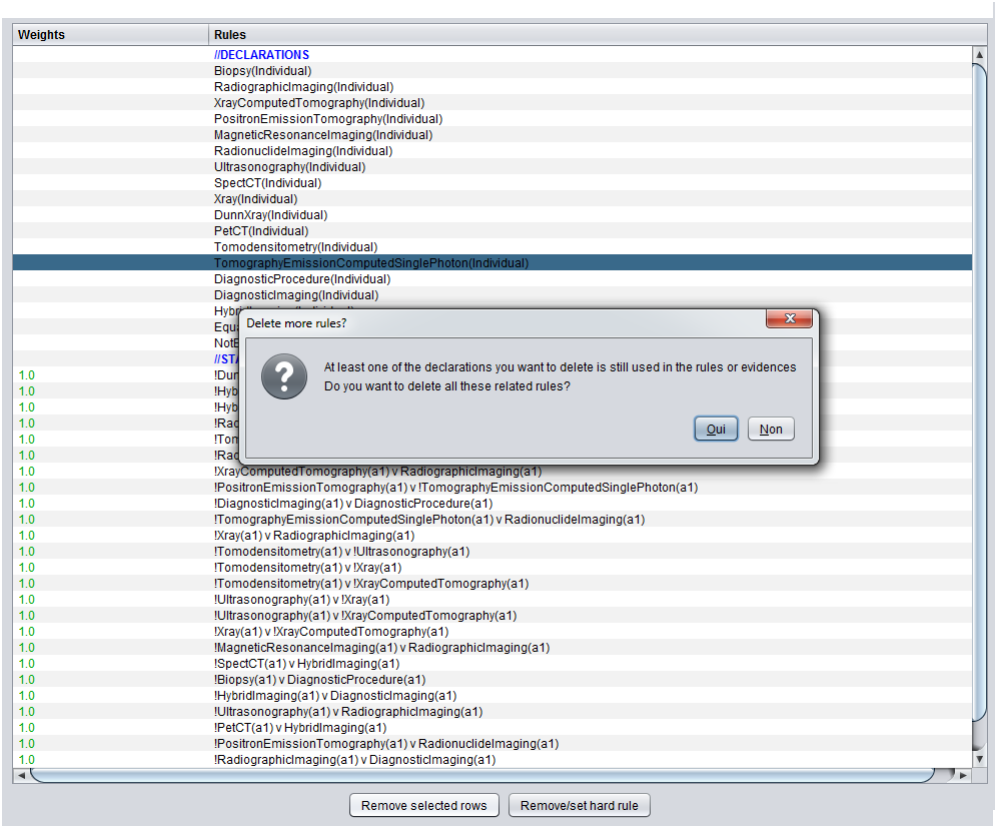


FIGURE A.8 – Suppression de déclarations toujours utilisées (3)

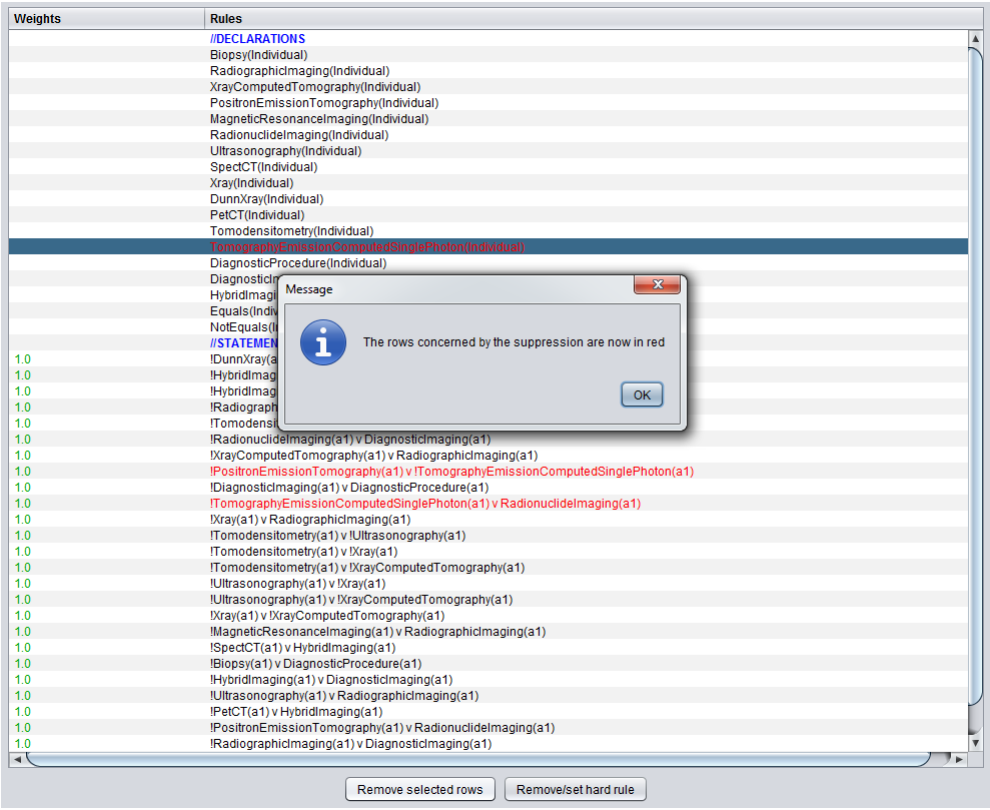
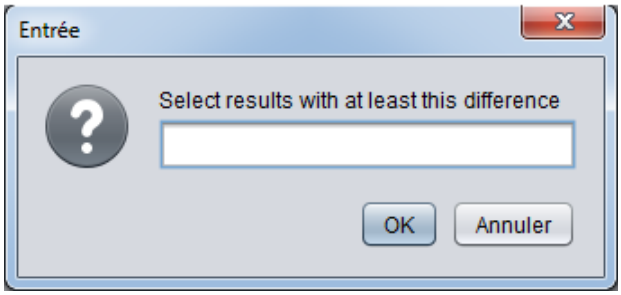



FIGURE A.9 – Statistiques - Choix de la différence minimale pour la comparaison de résultats



## A.3 Annexe de la partie d'expérimentation

FIGURE A.10 – Rapport entre le temps d'exécution et le nombre de règles pour Alchemy

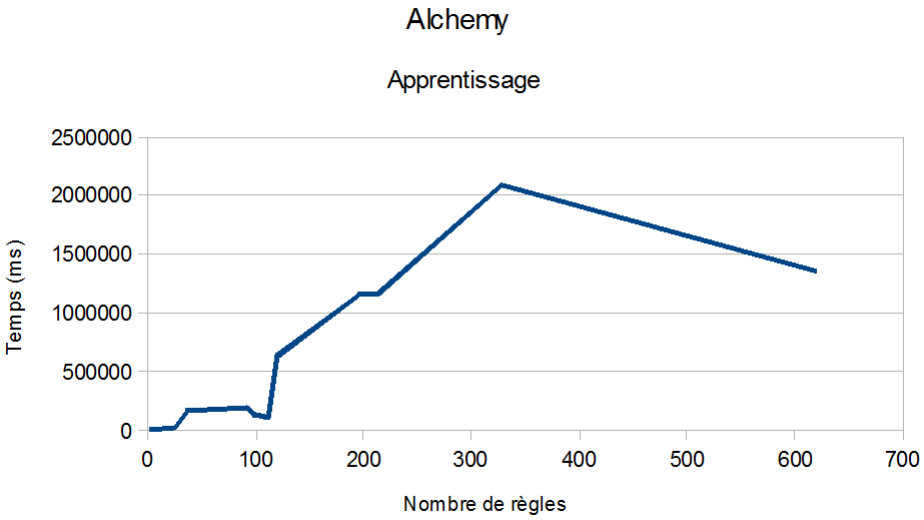
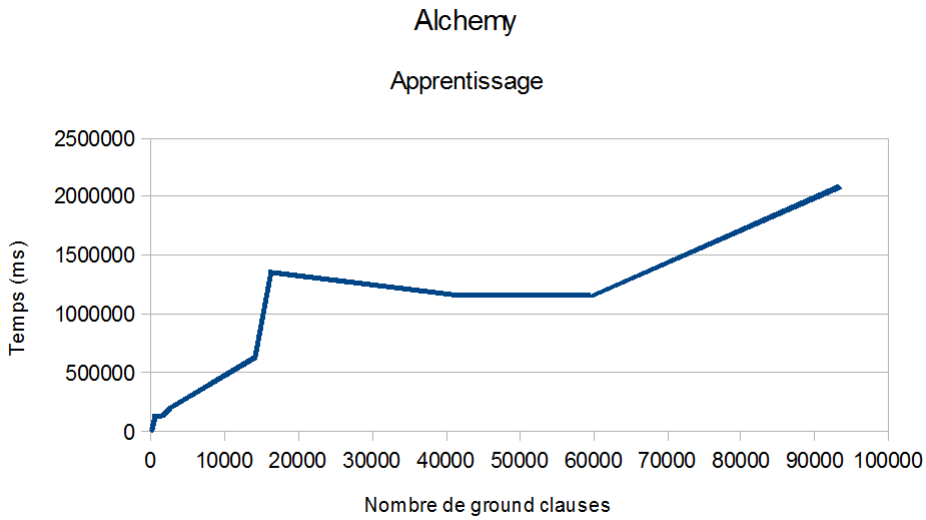


FIGURE A.11 – Rapport entre le temps d'exécution et le nombre de ground clauses pour Alchemy



# Bibliographie


[1] James Ortiz Axel Bodart, Keyvin Evrard and Pierre-Yves Schobbens. Arthur : A tool for markov logic network. *10th International IFIP Workshop on Semantic Web and Web Semantics*, October 2014. University of Namur, Computer Science Faculty, Belgium.

[2] Damien De Nizza, James Ortiz, Hubert Meurisse, Pierre-Yves Schobbens, and Jean-Paul Trigaux. Orthogen : Système d'information intégré pour la traçabilité et la gestion multi-paramètres des infections orthopédiques. 2013. Radiology Unit RDGN/CUMG, Catholic University of Louvain and Computer Science Faculty, University of Namur.

[3] Damien De Nizza, James Ortiz, Hubert Meurisse, and Pierre-Yves Schobbens. Formalisation et construction d'une ontologie dans le domaine des infections orthopédiques. 2013. Radiology Unit RDGN/CUMG, Catholic University of Louvain and Computer Science Faculty, University of Namur.

[4] Matthew Richardson and Pedro Domingos. Markov logic networks. 2006. University of Washington, Department of Computer Science and Engineering, Seattle, WA 98195-250, USA.

[5] Vangie Beal. Ai - artificial intelligence. 2003. En ligne sur le site de Webopedia, http ://www.webopedia.com/TERM/A/artificial_intelligence.html.

[6] Matthew Horridge. A practical guide to building owl ontologies using protégé 4 and co-ode tools. 2009. The University Of Manchester.

[7] Alexandre Serres. Introduction à l'indexation. 2003. En ligne sur le site de l'Université Rennes 2, http ://www.sites.univ-rennes2.fr/urfist/Supports/Indexation/Indexation2Defis.html.

[8] Damien Massé. Logique et intelligence artificielle (licence 3). Université de Bretagne Occidentale, France.

[9] Marie-Laure Mugnier. Bases de la logique du premier ordre. 2012. Université Montpellier II – L3, France.

[10] Denis Lugiez. Logique du premier ordre. Université de Provence, France.

[11] Jean-Pierre Jouannaud. Logique du premier ordre. 2006. Ecole Polytechnique de Palaiseau, France.

[12] Adrien Pain et Philippe Morignot. Système à base de règles. 2013. EPITA - École De L'Intelligence Artificielle, France.

[13] Jean-Marie Jacquet. Techniques d'intelligence artificielle. 2014. Université de Namur, Faculté d'Informatique, Belgique.

[14] Shiow yang Wu. Service-oriented computing, lecture 5 : Semantic web. National Dong Hwa University, Department of Computer Science and Information Engineering, Taïwan.

[15] Philippe Thiran. Web technologies, part i : Xml technologies. 2014. University of Namur, Computer Science Faculty, Belgium.

[16] François Lawarrée. Recherche de documents à partir d'ontologies de domaines. 2010. Université de Namur, Faculté d'Informatique, Belgique.

[17] Philippe Thiran. Web technologies, part iii : Semantic web technologies. 2014. University of Namur, Computer Science Faculty, Belgium.

[18] Jeff Heflin. An introduction to the owl web ontology language. Lehigh University, Pennsylvania, USA.

[19] Deborah L. McGuinness and Frank van Harmelen. Owl web ontology language overview. 2003. En ligne sur le site de l'Université de Standord, http ://www.ksl.stanford.edu/people/dlm/webont/OWLOverviewMay12003.htm.

[20] Francis Bach. Modèles graphiques probabilistes. 2006. Ecole des Mines de Paris, Centre de Morphologie Mathématique, France.

[21] François Pérès. Modèles graphiques probabilistes. Ecole Nationale d'Ingénieurs de Tarbes, France.

[22] Olivier Parent et Julien Eustache. Les réseaux bayésiens - a la recherche de la vérité. 2007. Université Claude Bernard Lyon 1, France.

[23] L'équipe PR-OWL/MEBN/UnBBayes. Pr-owl : A bayesian extension to the owl ontology language. En ligne sur le site http ://www.pr-owl.org/.

[24] Sargur Srihari. Machine learning and probabilistic graphical models. University at Buffalo, New York, USA.

[25] Feng Niu, Christopher Ré, AnHai Doan, and Jude Shavlik. Tuffy : Scaling up statistical inference in markov logic networks using an rdbms. 2010. University of Wisconsin-Madison, USA.

[26] Quang Thang Dinh. Apprentissage statistique relationnel : Apprentissage de structures de réseaux de markov logiques. 2011. Université d'Orléans, Laboratoire d'Informatique Fondamentale d'Orléans, France.

[27] Guy Cohen. Convexité et optimisation. École Nationale des Ponts et Chaussées, France.

[28] Jean Charles Gilbert. Éléments d'optimisation différentiable : Théorie et algorithmes. 2007. Centre de recherche Inria, Paris, France.

[29] Parag Singla and Pedro Domingos. Discriminative training of markov logic networks. University of Washington, Department of Computer Science and Engineering, Seattle, WA 98195-2350, USA.

[30] Daniel Lowd and Pedro Domingos. Efficient weight learning for markov logic networks. Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195-2350, USA.

[31] Lukas Kroc. Introduction to markov chain monte carlo. Cornell University, Deptartment of Computer Science, New York, USA.

[32] Hoifung Poon and Pedro Domingos. Sound and efficient inference with probabilistic and deterministic dependencies. Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195-2350, USA.

[33] Aikaterini Mpagouli and Ioannis Hatzilygeroudis. A rule-based system implementing a method for translating fol formulas into nl sentences. University of Patras, School of Engineering, Department of Computer Engineering and Informatics, 26500 Patras, Hellas.

[34] Aikaterini Mpagouli and Ioannis Hatzilygeroudis. Converting first order logic into natural language : A first level approach. University of Patras, School of Engineering, Department of Computer Engineering and Informatics, 26500 Patras, Hellas.

[35] Marc Sumner and Pedro Domingos. The alchemy tutorial. 2010. University of Washington, Department of Computer Science and Engineering, USA.

[36] AnHai Doan, Feng Niu, Christopher Ré, Jude Shavlik, and Ce Zhang. User manual of tuffy 0.3. 2011. University of Wisconsin-Madison, USA.

[37] Hoifung Poon, Pedro Domingos, and Marc Sumner. A general method for reducing the complexity of relational inference and its application to mcmc. University of Washington, Department of Computer Science and Engineering, Seattle, WA 98195-2350, USA.

[38] Jan Noessner. Rockit : A fast markov logic solver. 2013. University of Mannheim, Mannheim, Germany.

[39] Jan Noessner, Mathias Niepert, and Heiner Stuckenschmidt. Rockit : Exploiting parallelism and symmetry for map inference in statistical relational models. University of Mannheim, Mannheim, 68131, Germany and University of Washington, Seattle, WA 98195-2350, USA.